NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

REAL-TIME EVENT EXECUTION MONITORING

by

John J. Drummond

September, 1997

Thesis Advisors: Man-Tak Shing
Valdis Berzins

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE September 1997.	3. REPORT TYPE AND DATES COVERED Master's Thesis

4. TITLE AND SUBTITLETITLE OF THESIS. Real-time Event Execution Monitoring	5. FUNDING NUMBERS
6. AUTHOR(S) John J. Drummond	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey CA 93943-5000	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.	12b. DISTRIBUTION CODE

13. ABSTRACT *(maximum 200 words)*

Currently the Computer Aided Prototyping System software development environment provides monitoring techniques for real-time tasking execution times. However, these techniques are constrained in that there is only a provision for simple error messages to be presented upon execution failure such as that caused by a missed deadline. This approach necessitates that the software system designer haphazardly guess a task set execution time.

This thesis performed an examination of fine grain execution timing. This work was accomplished through the development of a program to perform true dynamic run time data collection of the typical task set execution exhibited within a real-time environment.

The results of this work is an accurate and efficient real-time task set execution monitoring software program which assists in overcoming the problem of task set execution run time prediction. The program itself has been embedded within the Computer Aided Prototyping System environment and is an enhancement over the previous monitoring technique by providing the system designer with true and accurate run time execution times. The validation of the thesis work has been performed by successful design and development of time critical real-time prototype software within the Computer Aided Prototyping System using the execution monitoring program.

14. SUBJECT TERMS Real-Time, Execution Run Time, Prototyping, Event Monitoring	15. NUMBER OF PAGES 149
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18 298-102

Approved for public release; distribution is unlimited.

REAL-TIME EVENT EXECUTION MONITORING

John J. Drummond
B.S.C.S., San Diego State University, 1992

Submitted in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN SOFTWARE ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
September 1997

ABSTRACT

Currently the Computer Aided Prototyping System software development environment provides monitoring techniques for real-time tasking execution times. However, these techniques are constrained in that there is only a provision for simple error messages to be presented upon execution failure such as that caused by a missed deadline. This approach necessitates that the software system designer haphazardly guess a task set execution time.

This thesis performed an examination of fine grain execution timing. This work was accomplished through the development of a program to perform true dynamic run time data collection of the typical task set execution exhibited within a real-time environment.

The results of this work is an accurate and efficient real-time task set execution monitoring software program which assists in overcoming the problem of task set execution run time prediction. The program itself has been embedded within the Computer Aided Prototyping System environment and is an enhancement over the previous monitoring technique by providing the system designer with true and accurate run time execution times. The validation of the thesis work has been performed by successful design and development of time critical real-time prototype software within the Computer Aided Prototyping System using the execution monitoring program.

TABLE OF CONTENTS

LIST OF FIGURES AND TABLES

LIST OF SYMBOLS, ACRONYMS AND/OR ABBREVIATIONS

CAPS Computer Aided Prototyping System

PSDL Prototyping System Description Language

RM Rate Monotonic

RTOS Real-Time Operating System

COTS Commercial Off The Shelf

MET Maximum Execution Time

MRT Maximum Response Time

FW Finish Within

MCP Minimum Calling Period

EDF Earliest Deadline First

ESF Earliest Start time First

DEDICATION AND ACKNOWLEDGMENT

I would like to dedicate this work to Miss Anh Le, and Edward & Loretta Drummond who provided abundant moral support during the rough times. I would also like to acknowledge the support of my advisors, Dr. Valdis Berzins who generated the initial idea behind this work, and Dr. Man-Tak Shing who provided excellent guidance and direction throughout this endeavor.

I. INTRODUCTION

A. OBJECTIVE

When designing and developing embedded real-time systems, timing issues are of paramount importance. As such the problem of achieving strict real-time deadlines must be overcome to enable proper performance given a set of critically timed real-time tasks. By utilizing a rapid prototyping approach the designer of these systems is capable of performing an in depth examination of these timing issues and in turn can attempt to achieve accurate task set deadlines.

The run-time system monitor research discussed within this thesis is aimed at the development of feedback mechanisms for a typical system designer based upon the typical utilization of the Computer Aided Prototyping System environment for construction of real-time system prototypes.

The use of the Computer Aided Prototyping System in the area of real-time software design does allow for this in depth examination of timing issues. However, in the CAPS environment there exists a problem with the monitoring of real-time task sets during execution time. While there is an acute requirement to collect run-time statistics during the actual execution of real-time prototype software, at the same time effort must be focused on minimizing any excessive computational overhead which may result.

The previous method for observing the real-time task execution times within the Computer Aided Prototype System is that of presenting error messages upon the event of scheduling in accuracy. Utilization of this method is severely restricted by only having the capability of error message passing and not a more sophisticated run time data analysis approach. This current method forces the real-time system designer to rely solely on the haphazard guessing of the execution times and time budgets. This early monitoring work produces error messages only when previously declared deadlines are missed. These messages are also limited in that they do not report how much time passed beyond the deadline, additionally they do not provide any information if the computation finishes consistently ahead of schedule.

The Real-Time Event Execution Monitoring System Project will focus upon equipping the software designer with an improved and more logical analysis approach for constructing real-time task sets. This will be accomplished by performing accurate and timely measurement of real-time operator run time execution. This information will then be provided to the designer for the purpose of developing increasingly accurate and correct real-time software.

B. METHODOLOGY

The methodology which this thesis effort follows is outlined in three stages. First stage is the focus upon the development of a system which will generate a report containing the real-time measured execution time for each specific operator within the CAPS timing scope. This initial phase includes the software design, development, implementation, and testing of the Real-Time Event Execution Monitoring System program.

Once this has been accomplished the second stage is to the restructure the existing CAPS scheduler. This work will enable a running tabulation of task set execution times to be compiled. This phase will entail the integration of the Real-Time Event Execution Monitoring System into the CAPS scheduler module while keeping minimal impact on environmental overhead. This phase may also include the performance testing on this integration work.

Lastly the final stage of this thesis work will continue with the construction of a method to transmit this previously tabulated timing information to the application designer for utilization in the CAPS tool users prototype design effort. This final phase entails utilization of the file input/output Ada modules as well as leveraging off the existing CAPS I/O routines and procedures.

C. BENEFITS OF STUDY

The resulting work within the CAPS program will enable the assignment of execution time requirements to be handled more efficiently and with increased accuracy during the prototyping process. Additionally this research work will provide real-time measurements to ensure that critical tasking will be analyzed correctly and allow increased accuracy for real-time prototyping designs.

The previous method of utilizing CAPS for real-time prototype system development in which the determination of a given task set execution time relies upon the best estimate of the system designer will no longer be needed during the scheduling phase. Instead the design effort will utilize an accurate record of actual task set run time execution data for timing requirements of the prototype system.

This feedback to the system designer follows the rapid prototype paradigm by providing critical program support information back to the user in a timely manner. This timelines is a strategic factor in development of a prototype as noted by [1] "The goal of a prototype is different than that of a production software system. Efficient use of designer time and rapid feedback are more important than robust operation, efficient use of machine resources, and completeness."

D. SCOPE

The scope of the thesis is the design, development, and evaluation of an accurate execution time measurement facility for the CAPS system. In the performance of this mission and within this scope this thesis the attempt to answer three primary questions are explored.

1. Can run-time statistics be collected during the execution of a real-time prototype without imposing an excessive computational overhead?

2. Is the monotonic clock of Ada 95 sufficiently accurate to assess timing properties in adequate detail to support real-time systems design?

3. Is the variance in running times significant?

E. THESIS ORGANIZATION

The first chapter of this thesis serves as the project introduction. Chapter II provides the background in the areas of similar research work, the Computer Aided Prototyping System, and Real-Time issues. Chapter III talks about the requirements for building the Real-Time Execution Monitoring System, including the system goals and constraints. Chapter IV explores the design of the Real-Time Execution Monitoring System including system architecture, subsystem analysis, object interaction and grouping. Chapter V presents the implementation considerations including design decisions. Chapter VI includes the thesis conclusion and discussion about ongoing future research work. The Appendices contain the project software design documentation and source code, use case sheets, operation sheets, event listings, and code mapping outline.

II. BACKGROUND

A. RELATED WORK

This section examines research work related to run time monitoring. A large quantity of the research work noted here is indirectly (Vs directly) related to the Real-Time Event Execution Monitoring System project. The main focus of these research endeavors are not necessarily the measurement and analysis of real-time task set execution. Rather, their run time monitoring efforts are, in some cases, the byproduct of other research area focus, such as program monitoring Vs task level monitoring, parallel Vs non-parallel systems.

Early research work in the area of execution measurement can be found in [2]. Although not performed within a real-time environment, this work examines specific language based software at the program unit level. This Ada based work produced a set of tools for analysis called the Ada Test and Evaluation Tools (ATEST). The tool set includes packages for: Source Instrumenter; Path Analyzer; Automatic Path Analyzer; Performance Analyzer; Self-Metric Instrumentation and Analysis; and a Symbolic Debugger. The limitation of this program run time analysis work is defined by the Ada environment as noted in [2] "The purpose of these tools is to test and evaluate programs written in the Ada language." Specific details of an Ada program run time execution can be acquired using the ATEST as follows. Initially the Source Instrumenter performs a parsing of the Ada program to be analyzed and breakpoints are inserted within the program units to allow a Run Time Monitor access. Next, the Run Time Monitor records the execution data of the program. Lastly, the Report Generator produces a file containing the execution data. The Performance Analyzer produces a report for user evaluation which contains program execution timing information in net time and cumulative time listings.

The work of [3] focuses upon real-time embedded systems with the development of a timing analysis tool. The Assist in File-Tuning of Embedded Real-time systems (AFTER) tool provides an interactive analysis and scheduling prediction tool. This tool elaborates upon task set scheduling theory and monitoring research as well as design work

in real-time operating systems. This instrument allows the designer to perform real-time scheduling analysis after the implementation phase of the software design effort.

The AFTER approach initially collects raw timing data from the targeted real-time embedded system. The data is then analyzed to provide a temporal implementation image which pinpoints existent and/or likely timing difficulties. Scheduling predictions can then be obtained which are based upon fine adjustments to the software timing properties.

The AFTER system is comprised of four principal modules: the Data Collection & Storage Unit; the Filter Unit; the Analysis Unit; and the Parameter Modification Unit. The first module, Data Collection & Storage interfaces directly with the targeted system. Its main task is to acquire data from system parameters. The method of data collection is not specific although the preferred technique is the utilization of a profiling mechanism embedded within the operating system, source code instrumentation is also acceptable. This data is then forwarded to AFTER through a predefined filter module. The Filter module extracts raw data from the Data Collection & Storage Unit. The raw data may consists of task execution times, task period, interrupts times, and operating system overhead. This filtered information is analysis dependent and user selectable. This filtered output is then forwarded on to the Analysis Unit which is the essence of the AFTER system. The Analysis Unit operates in two modes: the predictor mode; and the analysis mode. When operating in the analysis mode, a temporal image is developed which is based upon the real data collected from the targeted system. Mathematically modeled schedulability analysis is then performed on the temporal image based upon the scheduling algorithms found within the targeted system. It is this phase that difficulties within the targeted systems real-time software are accentuated. Missed deadlines, critical task set overutilization, CPU locking are some of the areas which can be highlighted and presented to the user. The predictor mode utilizes read data and estimated data as the basis for predicting system timing characteristic changes attributed to fine tuning the targeted system. The changes in question can be implemented within the targeted system and perform a follow-up analysis to examine possible system improvements. The Parameter Modification Unit allows for the user to perform modifications upon the targeted system. These parameters include: the exchange of aperiodic server interrupt handlers; the

alteration of task frequency; task execution time changes; functionality changes: and switching task scheduling from dynamic to static.

Two algorithms for insertion of profile monitoring source code into programs were presented by [4]. This research effort utilizes the approach of measuring a program at the basic block level for instruction set utilization of the computer system programs. A profiling algorithm is used to optimize the placement of counters within the program to be profiled. A tracing algorithm is used to trace a sub sequence of the basic block which is optimized for the program's execution length. The results of utilizing the profiling and tracing algorithms is that of reducing the number of counters by a factor of two and reducing the file size and overhead by 20-40% respectively. Since this thesis deals with the execution instead of the block level instruction set these algorithms cannot be applied directly to this thesis.

To provide real-time program development assistance [5] describes an approach for monitoring execution timing information within the application. The behavioral effect of the run time monitoring intrusion into the targeted system on the program performance is examined by defining time as an element which is composed of two entities. The first entity is called "clock time" which represents actual time, the second entity represents the run time execution monitoring execution time called "intrusion time." This work defines the local time during the execution of task (instrumented) at a given site S as the value pair (CT, Δ). Here CT is the clock time of S, while Δ is the current intrusion time of S. The estimation of clock time at S during execution of the targeted program (uninstrumented) then is $CT - \Delta$. As shown with an illustration from this work in Figure 1 an example task T_1 starts execution at, time = 0, finishes execution at time = 11, with 3 units of monitoring intrusion. Here local time at the start T_1 is (0,0) and at the conclusion of T_1 is (11,3) which translates to execution time of 8.

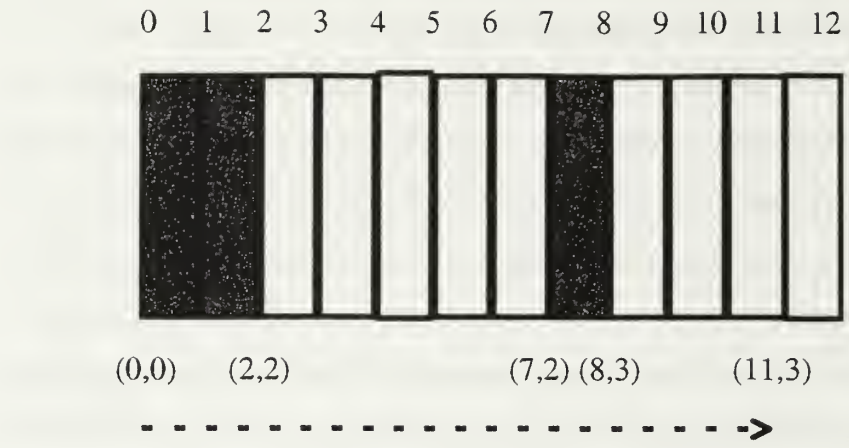

Figure 1. Run Time Monitoring Intrusion.[5]

This approach goes beyond measurement of tasks which finish within deadlines by allowing for measurement of real-time tasks which exceed deadlines.

In the domain of run time monitoring research there are specific differences between non-parallel and parallel computer systems. Within the parallel computer system inherent non-determinism and multiple threads of control are the most prominent. However, despite these differences insight from this parallel computer system run time research can increase understanding of overall run time monitoring techniques. The work of [6] in the area of run time monitoring research of parallel computer systems discusses the phases of system observation which can also be applied to non-parallel work. The investigation of system performance is segmented into three phases. The first phase includes the generation of run time data from system observation. Phase two consists of transmission and storage of the previously generated run time data. The final phase is that

of interpretation and user utilization of the run time data as shown in their illustration Figure 2.

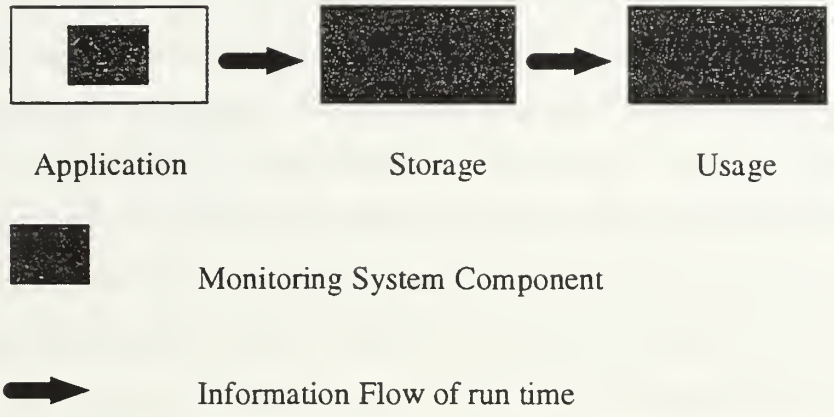

Figure 2. Three phases in gathering and using run time informatnions. [6]

B. THE COMPUTER AIDED PROTOTYPING SYSTEM

The Real-Time Event Execution Monitoring System program will reside in the execution support subsystem of the Computer Aided Prototyping System(CAPS). The main motivation being the previous release of CAPS of an adequate existing technique for run time execution monitoring within the CAPS environment. Another factor was the modular design of the CAPS real-time environment, which allowed for a less constrained interface design for the Real-Time Event Execution Monitoring System. The eventual targeting of the CAPS environment provided an ideal locale for the Real-Time Event Execution Monitoring System software.

The Computer Aided Prototyping System is an environment which provides system designers with real-time software development tools for constructing large scale systems with hard real-time constraint requirements. "The Computer Aided Prototyping System is an integrated software development environment aimed at rapidly prototyping

hard real-time embedded software systems, such as missile guidance systems, space shuttle avionics systems, and military Command, Control, Communications and Intelligence (C3I) systems." [7] As its name implies, the CAPS environment follows a prototype methodology for software development process.

Within this automated environment, the system designer has the capability to traverse the software development process beginning with requirements analysis then proceeding to code generation, advancing to subsystem integration, and after testing/debugging concluding with prototype demonstration and deployment for further analysis and implementation. As illustrated in [8] the CAPS environment allows a system designer to develop real-time systems through an iterative rapid prototyping process as shown in simplified form inFigure 3.

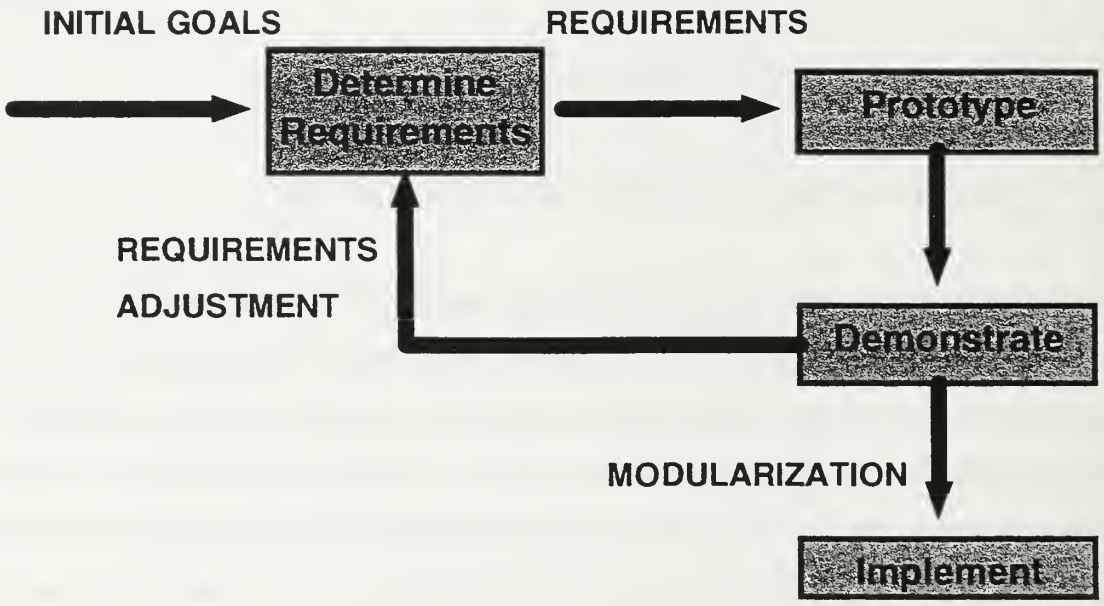

Figure 3. Example of Prototyping Process

The CAPS environment utilizes numerous editors including the Prototyping System Description Language (PSDL) editor, as well as an Ada editor with which the system designer can specify the prototype design and Ada source code.

Real-time scheduling of the prototype software is provided by the CAPS task scheduler, and provides for hard real-time as well as non-time-critical task sets. The CAPS real-time scheduling facility allows for both static and dynamic task scheduling. The static scheduling of hard real-time tasks is performed a priori based upon the system designer choice of Earliest Deadline First, Earliest Start time First, Bounded Backtrack, and Simulated Annealing scheduling algorithms. The dynamic scheduling of non-time-critical tasks takes place during run time.

An integral component of the CAPS environment is the high-level language PSDL. This language provides a mechanism for real-time specification within the CAPS environment through the use of real-time constructs. The CAPS PSDL component allows for proper modeling of timing issues as well as control constraints developed within the prototype system by the system designer. As illustrated in "A Prototyping Language for Real-Time Software" [9] the formal PSDL computational model is an augmented graph:

$$G = (V, E, T(v), C(v))$$

where

V = set of vertices

E = set of edges

T(v) = maximum execution time for vertex v

C(v) = set of control constraints for vertex v

C. REAL-TIME ISSUES

The source code implementation of the Real-Time Event Execution Monitoring System project is based upon the Ada programming language. At the outset of this project

the original source code implementation was accomplished using the real-time Ada83 constraints. The project source code was later revised using the Ada95 libraries with real-time annex constructs.

One of the major differences between these two Ada versions (from within the focus of this thesis) is that of the real-time extensions found within the Ada95 language system. The extensions of concern here can be found in section D.8, Monotonic Time of the Real-Time Systems Annex. The improved granularity afforded by the new Real-Time System includes constructs for Nanoseconds, Microseconds, and Milliseconds. This type of fine measurement capabilities were not provided outright in the Ada83 system and this change assisted greatly in the task of performing accurate run time execution measurements within the millisecond range.

The CAPS environment model does not call for granularity levels greater than 1 millisecond. The Translator program within the CAPS environment performs a calculation round-up to 1 millisecond of all significant digits found within the prototype timing specification. With this circumstance in mind the Real-Time Event Execution Monitoring System will operate at the minimum of millisecond granularity level. The timing granularity of a given environment is also based upon other factors as noted by [10]. "A language implementation is limited by the actual time-keeping resources provided by the hardware, which are possibly filtered through an operating system interface."

To determine clock granularity the target platform operating system in conjunction with the Ada 95 monotonic time construct accuracy's were examined. The Ada 95 Clock exhibited granularity within the 500 microseconds to 600 microseconds range.

To provide improved system for the rapid prototyping of embedded real-time software development the CAPS environment was also rebuilt using the Ada95 real-time library constructs. The CAPS environment had been previously built from the Ada83 language system. This improvement will allow the user of the CAPS environment to develop real-time applications containing stringent timing critical constraints within a more appropriate setting. The focus of the execution monitoring will be upon the CAPS operator objects. As illustrated by [11] the operator can be described as a taxonomy of time critical and non critical classes shown below in Figure 4.

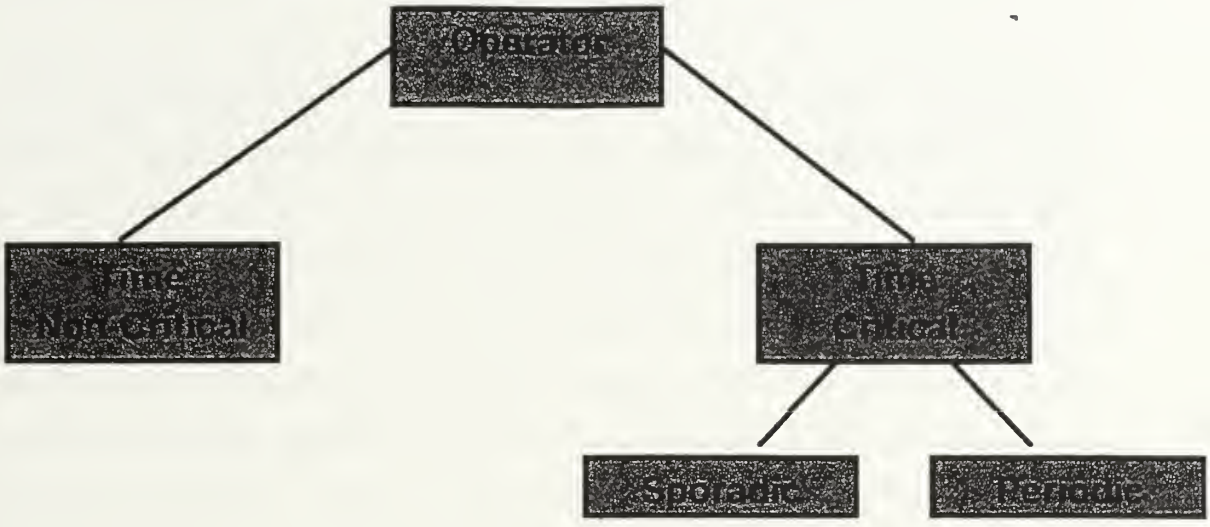

Figure 4 Operator Taxonomy

The Real-Time Event Execution Monitoring System will focus upon the actual execution times of the time-critical, statically scheduled operators. For the execution of a real-time task set to be accurately scheduled by the static scheduler within the CAPS environment the execution time must be assigned a priori. This constraint results in timing errors to occur as a result of a faulty choice of task set execution time assignments. The Real-Time Event Monitor System will provide information on real-time tasks which exceed their previously scheduled execution time, as well as those which underutilized their scheduled execution time. This tool provides the system designer with the data necessary to quickly isolate real-time execution requirements of a given task set. The Real-Time Event Execution Monitoring System will focus upon the analysis of these statically scheduled task sets within the CAPS environment, and no attempt will be made at this time to perform this analysis upon dynamically scheduled task sets.

III. REAL-TIME EVENT EXECUTION MONITORING SYSTEM REQUIREMENTS

This chapter provides an in-depth examination of the basic requirements specific to the Real-Time Event Execution Monitoring System endeavor. Within chapter III the system goals are summarized, functional requirements established, system constraints enumerated and responses to undesired events investigated and specified. The approach utilized in the Real-Time Event Execution Monitoring System development effort follows an Object Oriented methodology for real-time systems design. This approach is based upon the OCTOPUS methodology. OCTOPUS is a blending of both the Fusion and Object Modeling Technique methods. The requirements phase is the initial step towards the Real-Time Event Execution Monitoring System development.

A. SYSTEM GOALS

An initial Real-Time Event Execution Monitoring System project goal is to measure the real-time operators actual execution time. Additionally this project's other goals are to enable both hard and soft real-time deadlines within a given software design to be based on these measurements. Accomplishment of these elements will achieve the end goal of more efficient and effective use of the available computational resources in addition to providing for the meeting of strict real-time requirements of the operators.

Towards the achievement of the systems goals this project work will consist of the development of a system which will generate a report containing the measured execution time for each statically scheduled real-time operator within the CAPS timing scope. Additionally another Real-Time Event Execution Monitoring System goal will be to incorporate the restructuring of the existing CAPS environment to keep a running tabulation of maximum execution times.

These project goals will also include the transmission of this timing information to the system designer for utilization in the CAPS users prototype design effort. All of these objectives will be achieved through the analysis, design, and implementation of an accurate

execution time measurement facility for the CAPS system through utilization of the OCTOPUS design methodology.

The Real-Time Event Execution Monitoring System project has developed the following main goals for this work effort:

G1. The Real-Time Event Execution Monitoring System will measure real-time execution data for each CAPS real-time operator.

G1.1 The Real-Time Event Execution Monitoring System will keep a running tabulation of the operators actual execution times.

G1.2 The Real-Time Event Execution Monitoring System will provide for a restructuring of the existing CAPS environment to allow for pending tabulation of execution time data.

G2. The Real-Time Event Execution Monitoring System will analyze the CAPS operator run time data for correctness.

G2.1 The Real-Time Event Execution Monitoring System will compare the results of actual operator run time measured with planned run time.

G3. The Real-Time Event Execution Monitoring System will forward the CAPS operator run time data to the system designer.

G3.1 The Real-Time Event Execution Monitoring System will allow for utilization of CAPS real-time operator data in the CAPS tool users prototype design effort.

B. FUNCTIONAL REQUIREMENTS

The functional requirements for the Real-Time Event Execution Monitoring System have been based upon the examination of the initial problem statement* and the subsequently developed system goals. Additional information used in the functional requirements were assembled from the Real-Time Event Execution Monitoring System Use Cases found in Use case sheets Appendix A.

F1. The Real-Time Event Execution Monitoring System real-time timer shall be synchronized with the CAPS task schedule timer.

F2. The Real-Time Event Execution Monitoring System shall utilize the computer system clock as its main time source.

F3. The system clock used by the Real-Time Event Execution Monitoring System shall have timing with granularity fine enough to allow accurate run time measurement to be performed.

F4. The Real-Time Event Execution Monitoring System shall functionally operate timers based upon data from the system clock.

F5. The Real-Time Event Execution Monitoring System shall issue "start timer" prior to the CAPS real-time operator startup.

F6. The Real-Time Event Execution Monitoring System shall issue "stop timer" commands after the CAPS real-time operator stop times.

---

* Real-time event execution monitoring system Thesis Proposal 8/23/96

C. SYSTEM CONSTRAINTS

The system constraints for the Real-Time Event Execution Monitoring System have been compiled form information described within the initial problem statement* , the system goals, and functional requirements. Also, as was true in the development of the Functional Requirements, a significant information was assembled from the Real-Time Event Execution Monitoring System Use Cases located within Appendix A of this document.

C1. The Real-Time Event Execution Monitoring System must only be activated when a valid real-time operator has been created and is designated for monitoring.

C2. The Real-Time Event Execution Monitoring System must only initiate timer after the successful completion of Operator Run Time Measurement task.

C3. The Run Time Analysis task must only activate upon successful startup and completion of Run Time Measurement task.

C4. The transmission of operator run time data task must only be initiated upon successful startup, collection, and analysis of operator run time execution.

C5. The elapsed time between start timer request and timer start must be within 1 milliseconds.

C6. The Max time allowed from ReadOperatorStart and StartTimerExecute must be within 1 milliseconds.

C7. The Max time allowed from StartTimerExecute and GetClockReadout must be within 1 milliseconds.

---

* Real-time event execution monitoring system Thesis Proposal 8/23/96

C8. The Max time allowed from ReadOperatorStop and ReadTimer must be within 1 millisecond.

C9. The Max time allowed from ReadTimer and SendTimerData must be within 1 milliseconds.

D. RESPONSES TO UNDESIRED EVENTS

The Real-Time Event Execution Monitoring System design exposes the possibility of undesired events. This section examines potential undesired events and appropriate responses to their occurrence. The events have been identified based upon information within the preceding sections as well as the Real-Time Event Execution Monitoring System Use Cases which can be found within Appendix A at the end of this document.

U1. When the Real-Time Event Execution Monitoring System start timer command is not in concurrency with the CAPS real-time Operator startup the time delay must be counted and utilized in run time execution calculations.

U2. When the Real-Time Event Execution Monitoring System stop timer command is not in concurrency with the CAPS real-time Operator execution completion the time delay must be counted and utilized in run time execution calculations.

U3. When failure to read planned run time characteristics from a valid real-time operator occurs, an error message code must be transmitted to the run time analysis task.

U4. When timer read clock task fails to provide correct time measure, an error message code must be transmitted to the run time measure task.

IV. DESIGN OF THE REAL-TIME EVENT EXECUTION MONITORING SYSTEM

This chapter will serve as the formulation point of the design phase of the Real-Time Event Execution Monitoring System project. As mentioned in chapter III the OCTOPUS methodology will be followed for the Real-Time Event Execution Monitoring System development effort. Utilization of this OCTOPUS methodology within the design phase allows for better management of the heightened complexity present in the development of real-time systems while employing an Object Oriented approach which permits modular encapsulation. The components which comprise the Real-Time Event Execution Monitoring System design include and examination of system architecture, subsystem analysis and modeling, object interaction, event thread analysis, code mapping, and class specification.

Figure 5. Real-Time Event Execution Monitoring System (Simplified).

The results from the system execution of real-time operators will be viewed and monitored through the system designer user interface. The user interface is found in the CAPS environment. Shown in Figure 5 above is a collapsed diagram of this activity interaction.

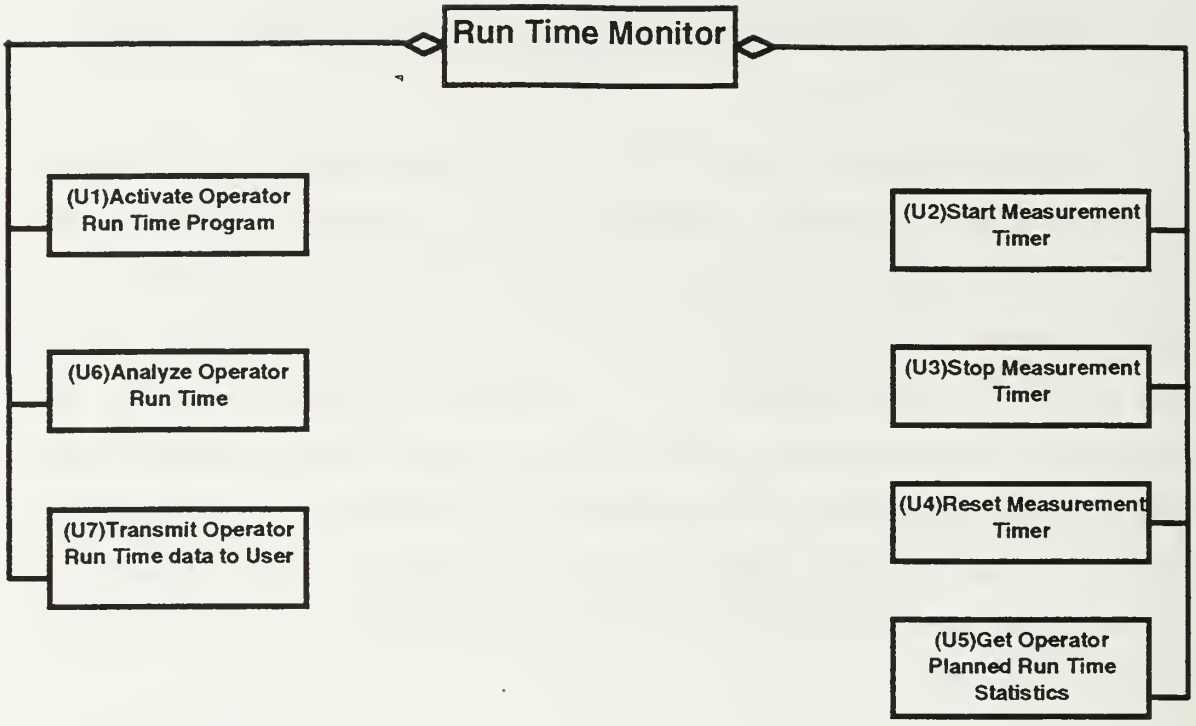

Figure 6 Real-Time Event Execution Monitoring System Use Case Diagram.

The Use Case Diagram as shown above in Figure 6 Real-Time Event Execution Monitoring System Use Case Diagram. describes a substantial portion of the design work in the development of Run Time Measurement System. This system is initially focused upon the Use Case Sheets, Use Case Diagram, and System Context Diagram. The individual Use Case Sheets can be found in Appendix A at the end of this document.

The system context diagram is shown below in Figure 7.

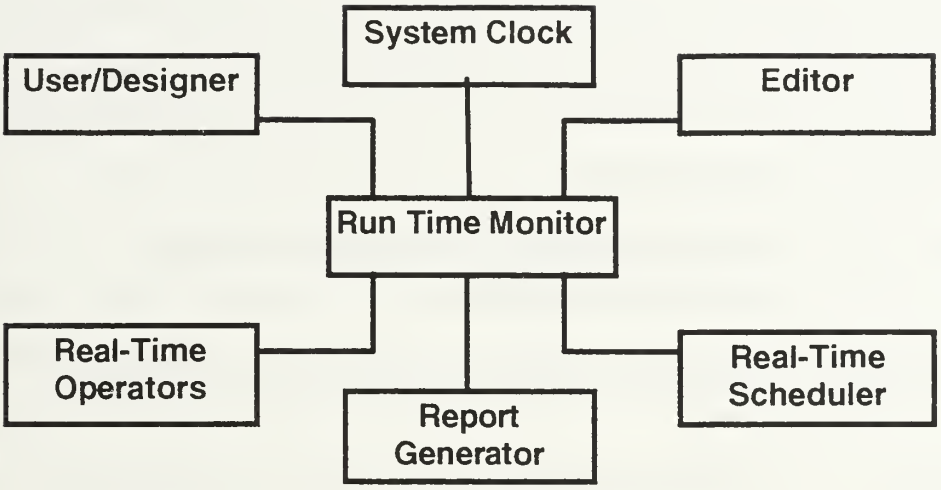

Figure 7. Real-Time Event Execution Monitoring System Context Diagram.

The context diagram for the Real-Time Event Execution Monitoring System was constructed from the Use Cases and based upon the Actors which will interact with the system. The system context diagram allows for a structural overview of the system environment.

The basis for this work was drawn from Table 1 example of the Use Case Sheet found in Object-Oriented Technology for Real-Time Systems.[12] Initially the actors for the Real-Time Event Execution Monitoring System Use Cases were determined by finding all external agents which would be interacting with the system. The Real-Time Event Execution Monitoring System external actors were then classified as Active/Passive, Client/Nonclient, Primary/Secondary, and roles were determined for each along with usage information based upon Table 1 below.

Use Case	**Name of case**
Actors	**External requesters of system services or autonomous activity**
Preconditions	**Conditions need to be satisfied to do the Use Case**
Description	**Short statement describing the Use Case (time req, exceptions, etc).**
Subuse cases	
Exceptions	**How the system responds to Exceptions listed above**
Activities	**Line test sequence**
Postcondition	**Conditions after the use case is successfully or unsuccessfully completed**

Table 1. Use Case Sheet Example.

A. SYSTEM ARCHITECTURE

To separate the task of analyzing the Real-Time Event Execution Monitoring System into reasonably modular components the system architecture has been partitioned. To accomplish this the Real-Time Event Execution Monitoring System has been

decomposed into various independent system domains. The resulting independent system domains include: User Interface Domain, Device Domain, Measurement Domain, and the Analysis Domain.

These domains will be representative of Real-Time Event Execution Monitoring subsystems and merged together later as part of the system integration task work. The activities being performed within these system domains represent different aspects of the entire system for example: the User Interface Domain will include System Designer, CAPS; the Device Domain will include System Clock functions; the Measurement Domain will be responsible for actual run time calculation; and the Analysis Domain will perform the function of comparative evaluation of the planned and actual operator run times which will be collected.

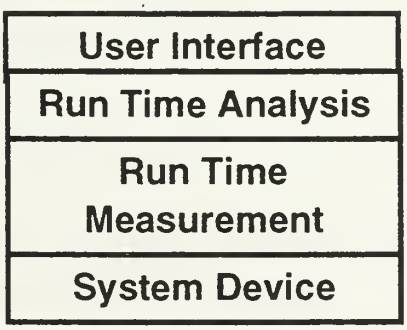

Figure 8. Real-Time Event Execution Monitoring System Architecture Layers.

As shown above Figure 8 illustrates the layered architecture resulting from the decomposition of the Real-Time Event Execution Monitoring System into specific independent domains. Communication between domains will be handled via formal parameter passing. This design will allow for loose module coupling.

The Real-Time Event Execution Monitoring System applications subsystems as well as hardware interface has been illustrated in the Subsystems Diagram shown below in

Figure 9. The Real-Time Event Execution Monitoring System application subsystems include: Run Time Analysis; System Device; Run Time Measurement: and User Interface.

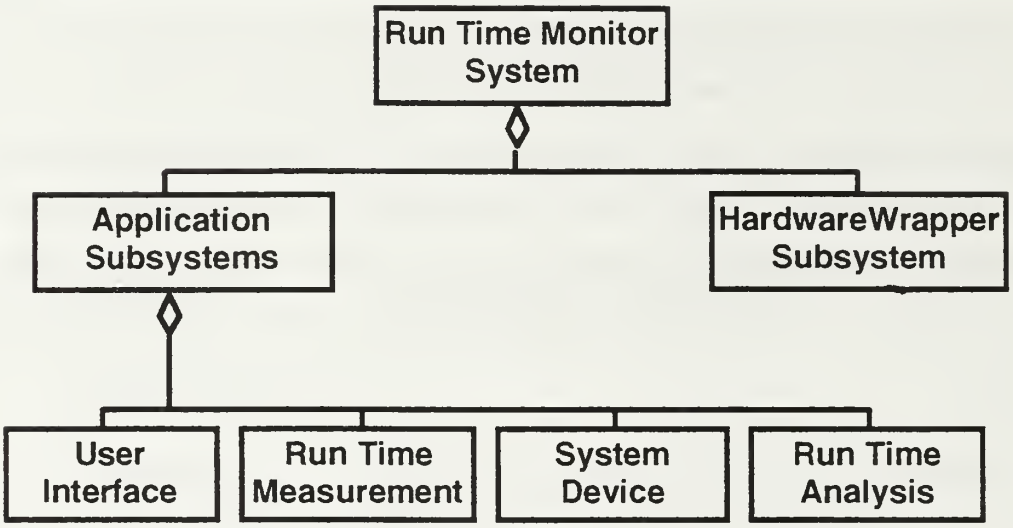

Figure 9. Run Time Monitoring Subsystems Diagram.

The interaction of these subsystems within the Real-Time Event Execution Monitoring System is shown by a command table found below in Table 2. A Shared Memory Access Table for the system has not been developed, as there will be no shared memory access across the Run Time Monitoring subsystems. Communication across subsystems will take place via formal parameter passing technique. Command intercommunication between these subsystems is represented by the command table below.

From	To	Command
User Interface	R/T Measurement	Start measurement
R/T Analysis	User Interface	Get Planned Data
R/T Analysis	R/T Measurement	Get Performance Data
User Interface	R/T Analysis	Analyze Data
R/T Measurement	System Device	Start Timer
R/T Measurement	System Device	Stop Timer
R/T Measurement	System Device	Reset Timer
System Device	R/T Measurement	Get Time
R/T Analysis	User Interface	Transmit Data

Table 2. Real-Time Event Execution Monitoring System Command Table.

The four main subsystems will rely upon communication shown in Table 2 above to transmit commands among subsystems. These commands for the Real-Time Event Execution Monitoring System will be passed between the respective subsystems via formal procedure calls and parameter passing as mentioned above.

B. SUBSYSTEM ANALYSIS

This section focuses upon detailed inspection of the requirements specification and application domain of the Real-Time Event Execution Monitoring System. Three Models are constructed during the Subsystem Analysis phase of software development: the Object Model, the Functional Model, and the Dynamic Model.

The Object Model includes the Real-Time Event Execution Monitoring System Class Description Table, and a Class Diagram which describes objects domain and relationships. The Functional Model is made up of Operations Sheets. These Operation Sheets provide a focus upon functional interface and services among Run Time Monitoring subsystems.

The Dynamic Model is comprised of Event Lists, Event Sheets, State Charts, Scenarios. The Event Lists, and Event Sheets focus upon the Real-Time Event Execution Monitoring System input occurrences such as real-time operator start. The State Charts allow for detailed analysis of the Real-Time Event Execution Monitoring System states within manageable complexity limits. The state charts for Run Time Measure, Run Time Analysis, Timer, and Run Time Results are included within this analysis. The Scenarios for the Real-Time Event Execution Monitoring System sequence of events include Run Time Measure, Run Time Analysis, and Run Time Results. These scenarios are included at the end of this document in Figure 16, Figure 17, and Figure 18.

The Real-Time Event Execution Monitoring System has been decomposed into four main subsystems as mentioned in Chapter IV including: User Interface; Run Time Analysis; Run Time Measurement; and System Device as shown in Figure 8. The major intercommunication between these subsystems is represented by the command table found in Table 2.

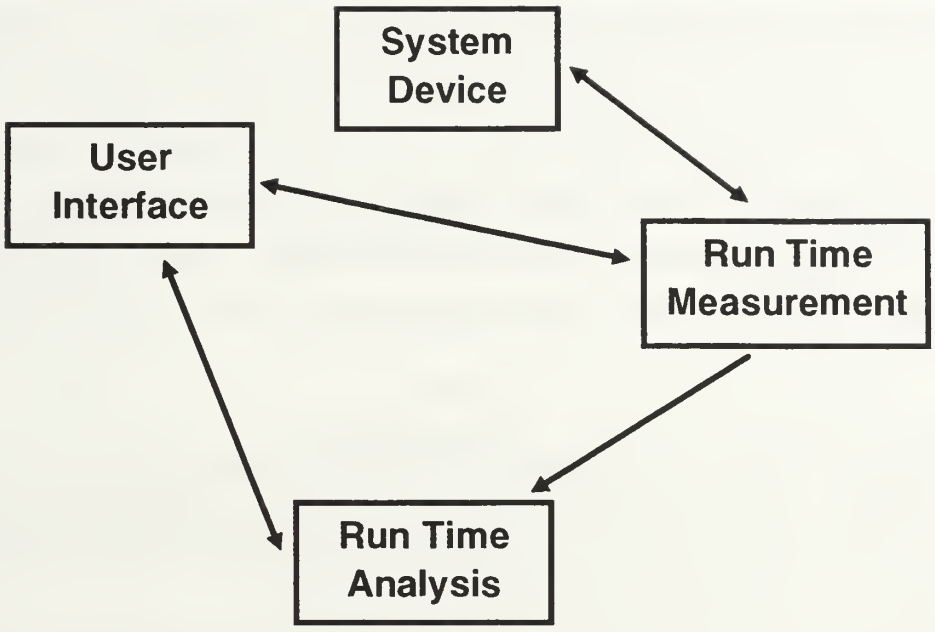

Figure 10 Run Time Monitoring Subsystem Interfaces.

The four main subsystems will rely upon the communication shown in Figure 10 above to transmit commands among subsystems. Formal parameters passed between these subsystems will allow for control and data information to be exchanged.

1. Object Model

The Real-Time Event Execution Monitoring System objects and relationships are described in this section. The development of the Object Model was derived through an iterative process. Initially the System Use Cases were examined, and scanned for all nouns which could represent objects within the system. A list of potential objects was then formulated from this first pass. Next the potential object list was compared with Use Case

objectives to determine object relevancy. The finalized object list for the Real-Time Event Execution Monitoring System was then completed.

The objects that are within the Application Subsystem Boundary include: Run Time Analysis, Timer, Run Time Measure, and Run Time Results. The objects which are outside the boundary include: Operator, CAPS, Clock, and System Designer.

From this Real-Time Event Execution Monitoring System Object list a Class Description Table was conceived. The Class Description Table can be seen below.

Class	Description
R/T Analysis	**Compare planned/actual run times**
Timer	**Count execution run time**
R/T Measure	**Sync timer with actual operator execution start/stop**
R/T Results	**Transmit run time analysis results to System Designer**
Operator	**Execution entity**
CAPS	**Prototyping execution environment**
Clock	**Real-time system measure device**
User	**Real-time development environment**

Table 3. Class Description Table for Real-Time Event Execution Monitoring System.

Using this information a Class Diagram for the Real-Time Event Execution Monitoring System was developed. This diagram shows the interaction between objects,

both within the subsystem boundary and outside the boundary. The Class Diagram shown below in Figure 11 indicates the segmentation between the application subsystem boundary.

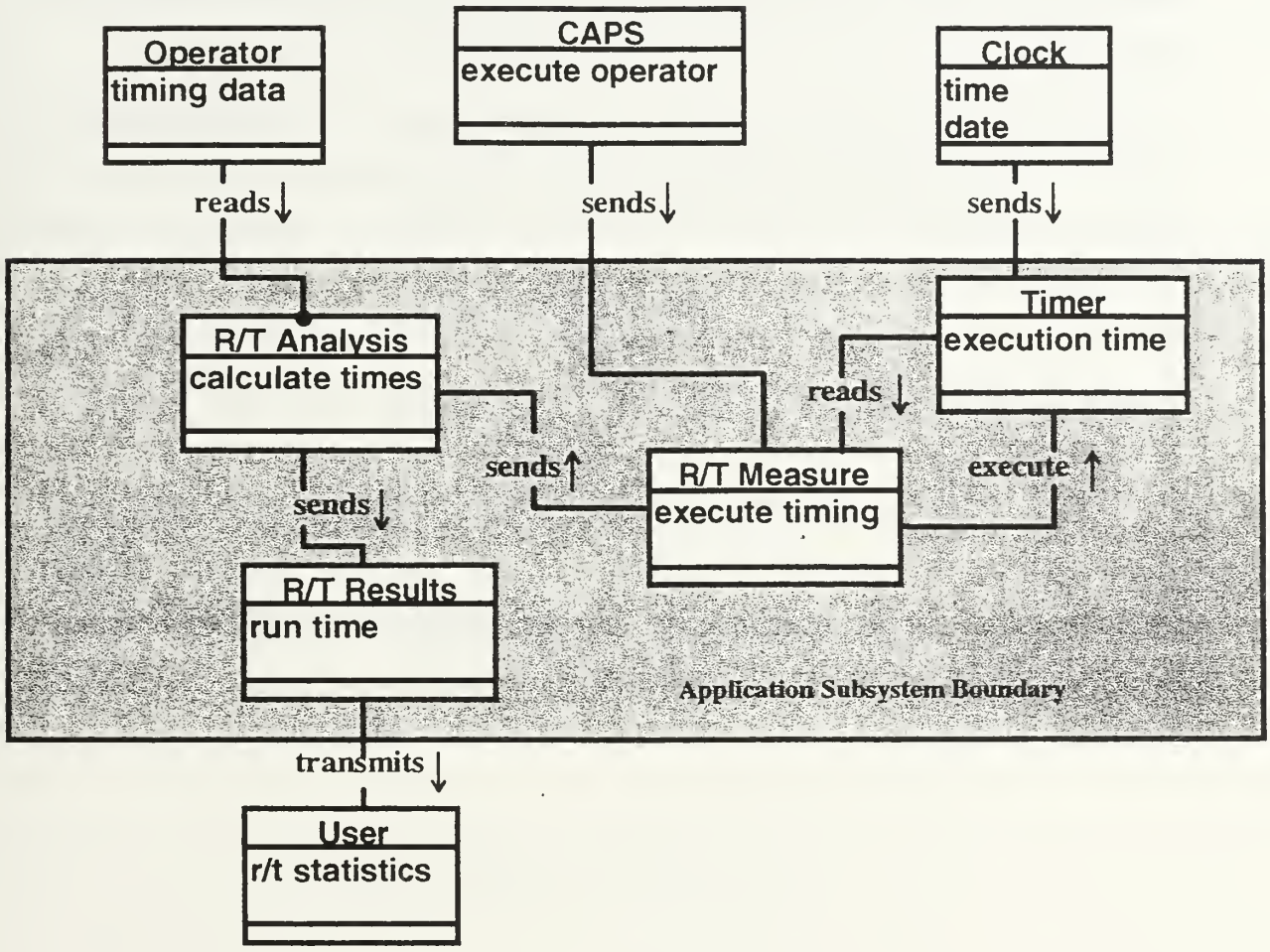

Figure 11. Class Diagram for the Real-Time Event Execution Monitoring System.

The correctness of the Object Model was checked by examining the previously developed Real-Time Event Execution Monitoring System Use Cases and verifying the interaction of the objects within the model in completing specific tasks.

2. Functional Model

The Real-Time Event Execution Monitoring System functional model is used to illustrate the functional interface of the subsystems. The Real-Time Event Execution Monitoring System has been decomposed into four main subsystems including: User Interface; Run Time Analysis; Run Time Measurement; and System Device.

The services provided by a given subsystem of the Real-Time Event Execution Monitoring System is depicted within the interface. The interfaces will be across the subsystem application boundary, between other subsystems and external agents of the Real-Time Event Execution Monitoring System. The functional model is described by Operation Sheets which represent the Real-Time Event Execution Monitoring System operations, associations, and other details. The Operations Sheets can be found in Appendix B.

The services provided by the subsystem as shown within the functional interface have been derived from examination of the Use Cases for the Real-Time Event Execution Monitoring System found in Appendix A. The Operation Sheets describing the functional model have been developed by following the example provided by [12] is shown below in Table 4.

Operation	Name of operation
Description	Short statement describing the operation
Associations	Associations to the classes and objects and possibly also to the events and s tates to which it is related
Precondition	Conditions that need to be satisfied to start the operation; these do not guarantee that the operation will be successfully completed
Inputs	Arguments that an operation needs to perform its desired function
Modifies	What modifications the operation causes on its arguments or on common data in the subsystem
Outputs	What information the operation needs to supply its client
Postcondition	Conditions after the operation is successfull y completed and the conditions that apply if the operation is terminated due to an error

Table 4. Example Operations Sheet.

3. Dynamic Model

The dynamic model for the Real-Time Event Execution Monitoring System represents an order of interaction between the system and the users of the system. Illustrating the possible operations within a specific state, the dynamic model outlines the resulting effect of the Real-Time Event Execution Monitoring System operations. The dynamic model also outlines the timelines of these specific operations and conditional

performance information the Real-Time Event Execution Monitoring System operations themselves.

The Dynamic model for the Real-Time Event Execution Monitoring System consists of initial analysis of events by creation of event lists and event sheets. The event sheet is a major component of the Real-Time Event Execution Monitoring System's dynamic model. The development of the Event Sheet is based upon the event sheet template proposed by [12] shown below in Table 5. Actual event sheets for the Real-Time Event Execution Monitoring System are listed in Appendix C beginning on page 72.

Event	**(E1) Name of the event**
Response	**The desired end-to-end response from the system**
Associations	**Associations to the classes and objects and possibly also to the related operations**
Source	**The originators of the event, for example, other subsystems or the hardware wrapper**
Contents	**Data attributes that an event may hold**
Response Time	**The maximum and minimum time limitations concerning the giving of the response**
Rate	**The rate of occurrence that can be, for example, at startup, periodic every 10 minutes, timed at 8:00 AM and 2:00 PM, occasional, exceptional, and so forth**

Table 5. Example Event Sheet.

Following the event analysis stage is the state analysis which is comprised of Real-Time Event Execution Monitoring System state charts found in the Figure 12. Timer State Chart, Figure 13. Run Time Measure State Chart, Figure 14. Run Time Analysis State Chart and, Figure 15. Run Time Results State Chart as shown below.

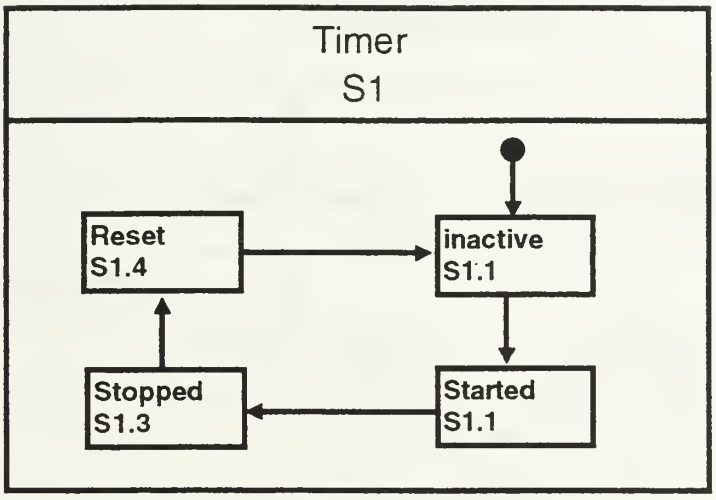

Figure 12. Timer State Chart

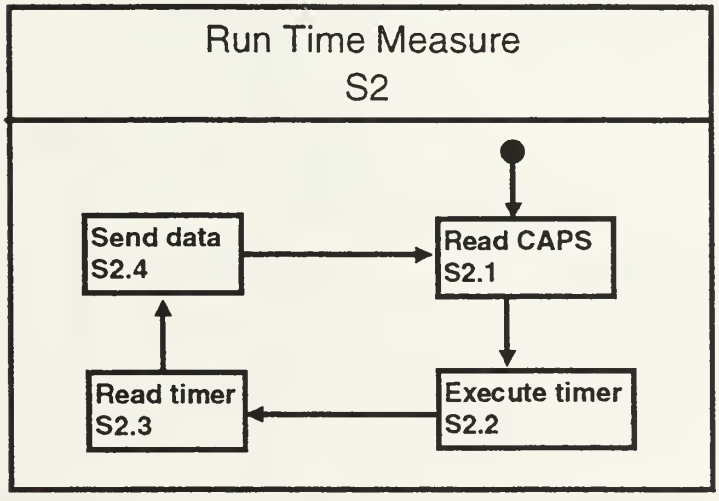

Figure 13. Run Time Measure State Chart

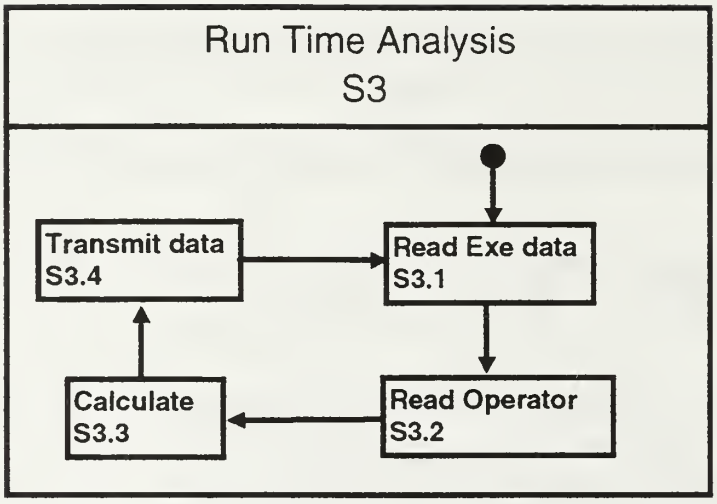

Figure 14. Run Time Analysis State Chart

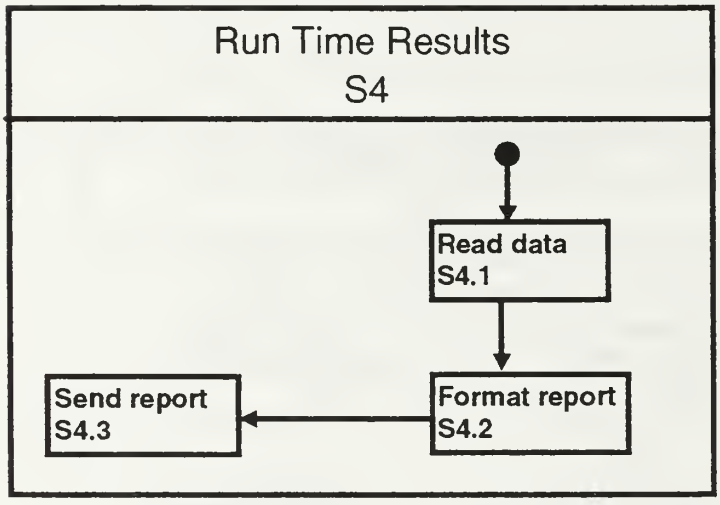

Figure 15. Run Time Results State Chart

The last stage of dynamic modeling for the Real-Time Event Execution Monitoring System is the actual validation of the model itself, by utilization of scenarios illustrated below in Figure 16, Figure 17, and Figure 18.

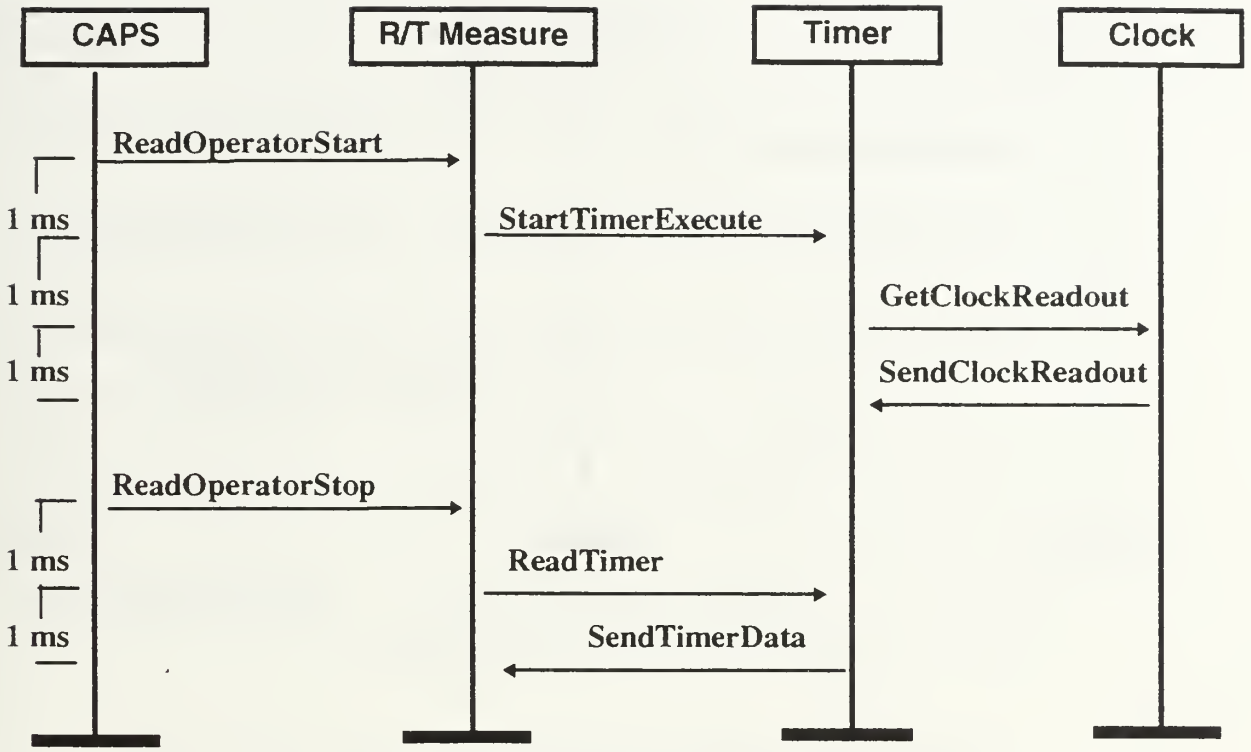

Figure 16. Run Time Measure Scenario.

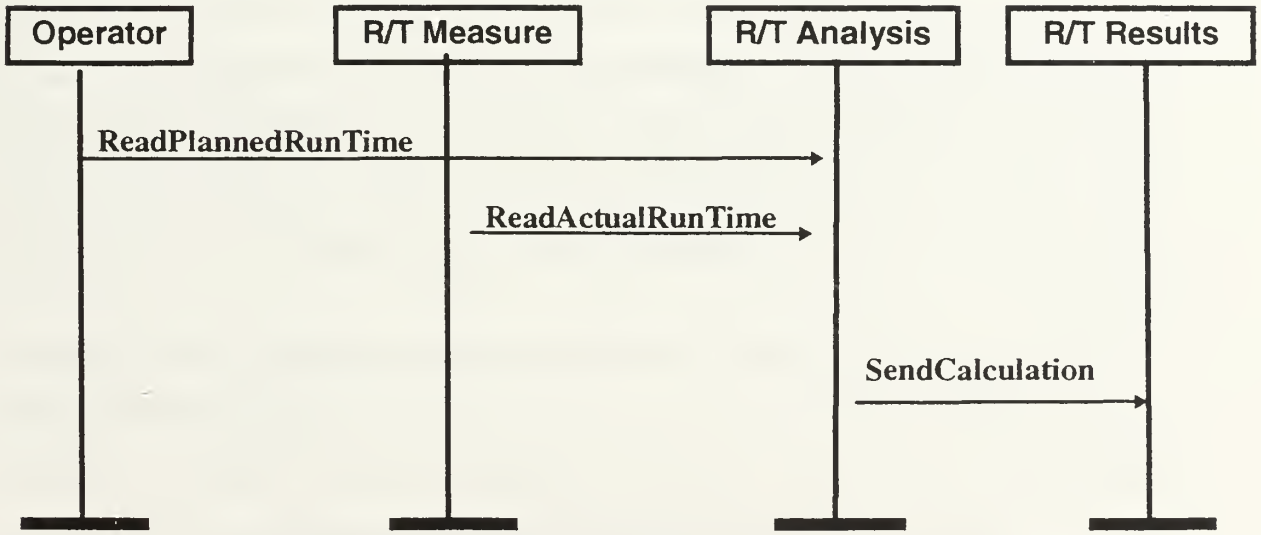

Figure 17. Run Time Analysis Scenario.

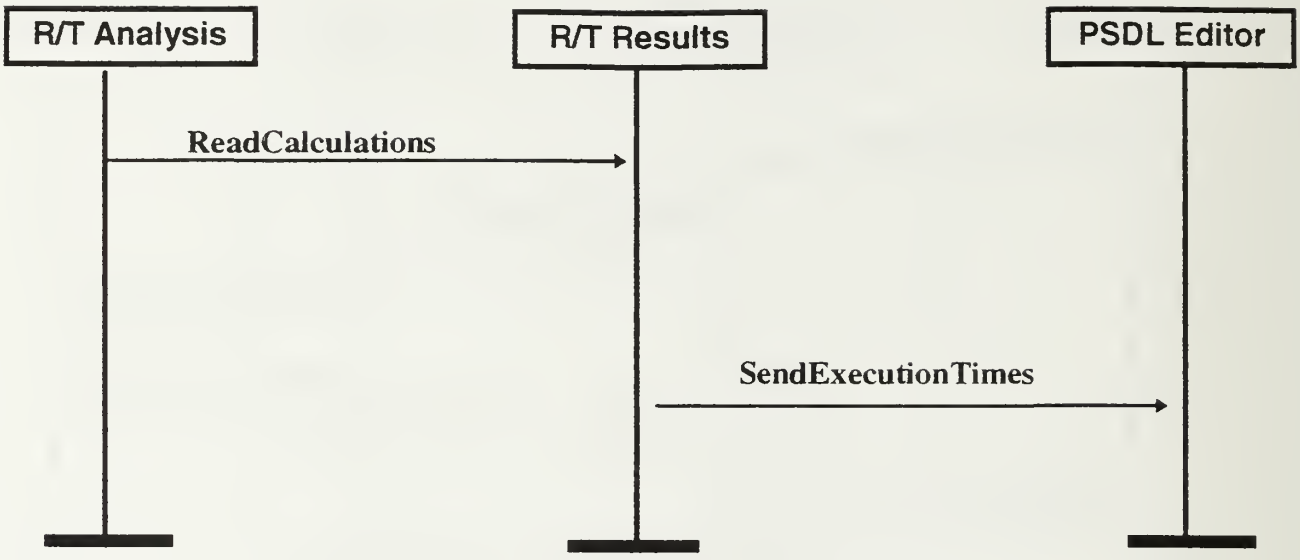

Figure 18. Run Time Results Scenario.

C. OBJECT INTERACTION

The Real-Time Event Execution Monitoring System project interaction between previously developed objects is outlined in this section. This object interaction is based upon the Event List, found in Table 4 below, encountered within the systems requirements section. The Event List was produced within the Dynamic Model of the Subsystem Analysis component which is comprised of Event Lists, Event Sheets, State Charts, Scenarios.

The Event Lists, in this case, will focus upon the Real-Time Event Execution Monitoring System input occurrences such as real-time operator start event. The object interaction from these input occurrences is detailed within Event Threads and Object Interaction Graphs of the Real-Time Event Execution Monitoring System. The Event List Table is shown below to provide coherence to this object interaction.

Event	Description
(E1) ReadOperatorStart	Get actual CAPS operator start time
(E2) ReadOperatorStop	Get actual CAPS operator stop time
(E3) ReadOperatorData	Get Planned operator execution times
(E4) GetClockTime	Get clock current readout
(E5) ReadTimerStart	Get timer start readout
(E6) ReadTimerStop	Get timer stop readout

Table 6. Event List for Real-Time Event Execution Monitoring System.

The initial task for describing object interaction was the construction of Event Threads. The Event Threads for the Real-Time Event Execution Monitoring System were developed by following the recommended [12] step-by-step approach for building event threads using the OCTOPUS method. This process is summarized in Table 7 below. The event threads created include: read operator start; read operator stop; get clock time; read operator data; read timer start; and read timer stop. The Event Threads are located in section E of this chapter.

1. Select an event not yet considered.
2. Identify an object involved in the processing of the selected event.
3. Design and record subsequent interactions.
4. Merge the new object interaction thread with existing ones.
5. Repeat from step 3 as long as other involved objects can still be found
6. Repeat from step 1 unless done for all events.
7. Iterate and balance the use of objects and statecharts.

Table 7. Building Event Threads for Real-Time Event Execution Monitoring System.

Next the development of Object Interaction Graphs were produced to provide a detailed representation of the Real-Time Event Execution Monitoring System object interaction. The development of these object interaction graphs is again based upon each of the previously created event threads within the Real-Time Event Execution Monitoring System. The object interaction graphs are also central to the lists of events derived from the system requirements development in Chapter III. The object interaction graphs are located in the next section.

D. INTERACTION GRAPH

The object interaction graphs created for the Real-Time Event Execution Monitoring System are based upon four major events. These events include: ReadOperatorStart; ReadOperatorStop; ReadOperatorData; GetClockTime; ReadTimerStart; and ReadTimerStop. The objects associated with these specific event

include objects outside as well as inside the Real-Time Event Execution Monitoring System application subsystem boundary.

The CAPS object and Run Time Measure object are both involved in the processing of the ReadOperatorStart and the ReadOperatorStop events and form the basis of the object interaction graph E1 and E2 respectively. Additionally statechart run time measure and statechart run time analysis are also involved in event processing included in the object interaction graph for the events. This can be seen in Figure 19 and Figure 20 below.

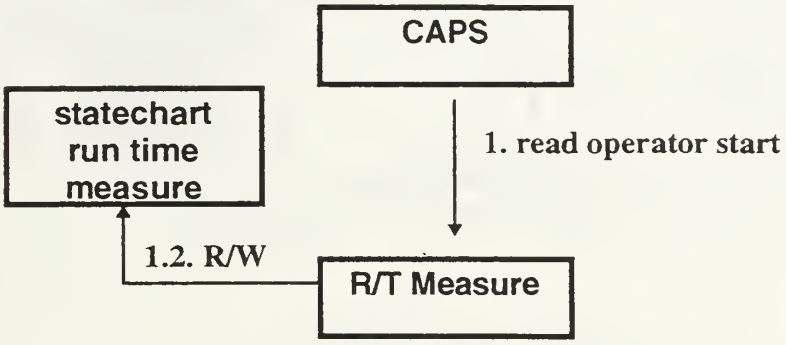

Figure 19. Object Interaction Graph E1

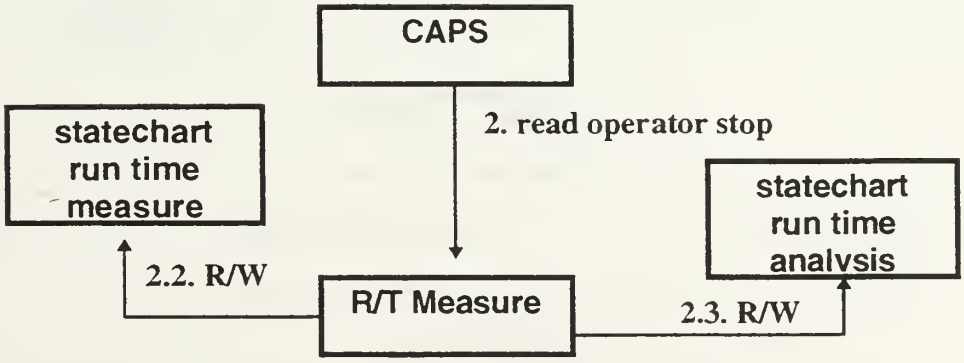

Figure 20. Object Interaction Graph E2.

The ReadOperatorData event is reliant upon the Operator object and the Run Time Analysis objects for processing as depicted in the object interaction graph E3. Also included within the object interaction graph E3 are the statechart run time results and statechart run time analysis.

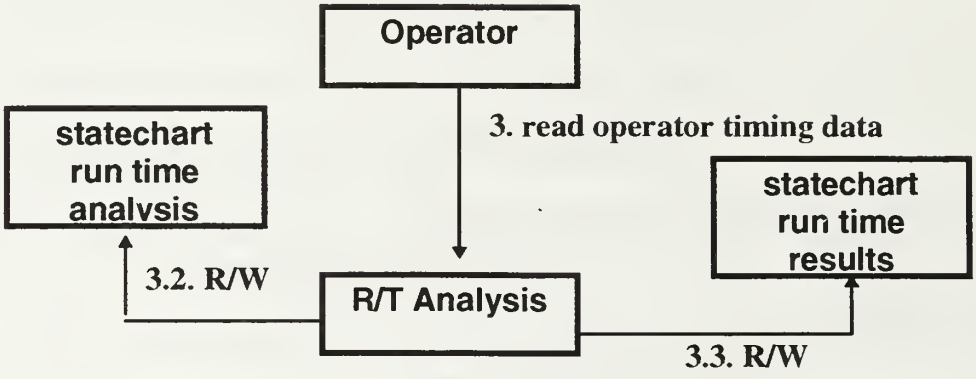

Figure 21. Object Interact Graph E3.

The object Clock and object Timer are directly involved in the processing of the GetClockTime event and are included within the object interaction graph E4. The statechart timer is also included in the E4 object interaction graph as it has some involvement in the processing of the GetClockTime event as shown below in Figure 22.

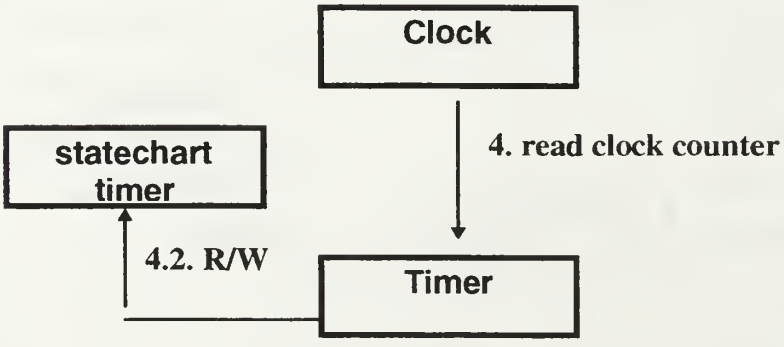

Figure 22. Object Interaction Graph E4.

The Timer object and Run Time Measure object are specifically involved in the processing of the ReadTimerStart and ReadTimerStop events as shown in object interaction graph E5 and E6. Observing Figure 23 and Figure 24 one can see that the statechart run time measure and statechart timer also take part in this event processing.

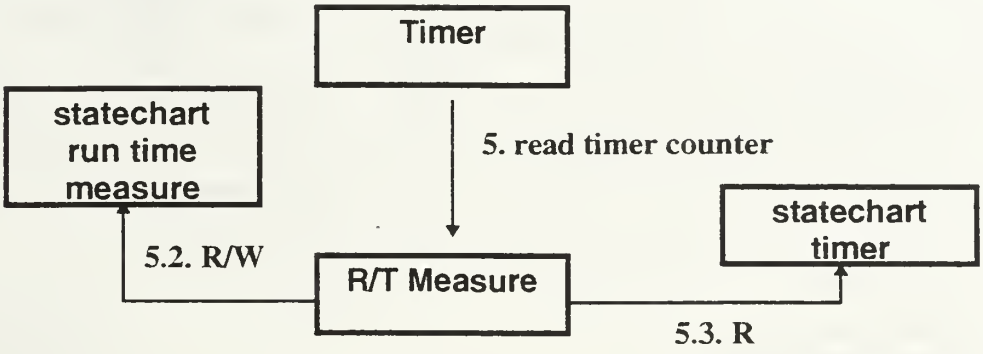

Figure 23. Object Interaction Graph E5.

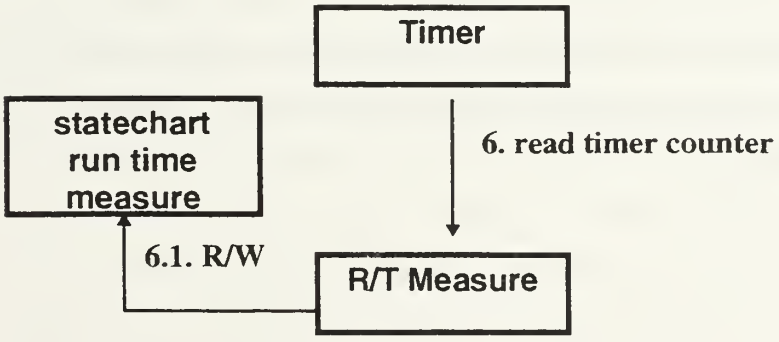

Figure 24. Object Interaction Graph E6.

The Run Time Analysis object and Run Time Results object together provide for the presentation of run time execution data to the user as shown in object interaction

graph E7. As illustrated in Figure 25 the statechart run time results is also included in this delivery of data to the user.

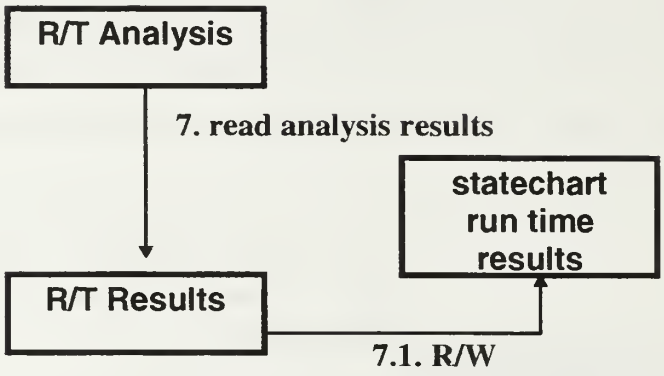

Figure 25 Object Interaction Graph E7.

E. EVENT THREADS

The event thread diagram for the Real-Time Event Execution Monitoring System includes the interaction of seven objects including: CAPS; Operator; Clock; Timer; Run Time Analysis; Run Time Analysis; Run Time Measure; Run Time Results; and four state charts which include: statechart run time analysis; statechart run time measure; statechart run time results; and statechart timer. This diagram illustrates all object interaction and can be found in Figure 26 below.

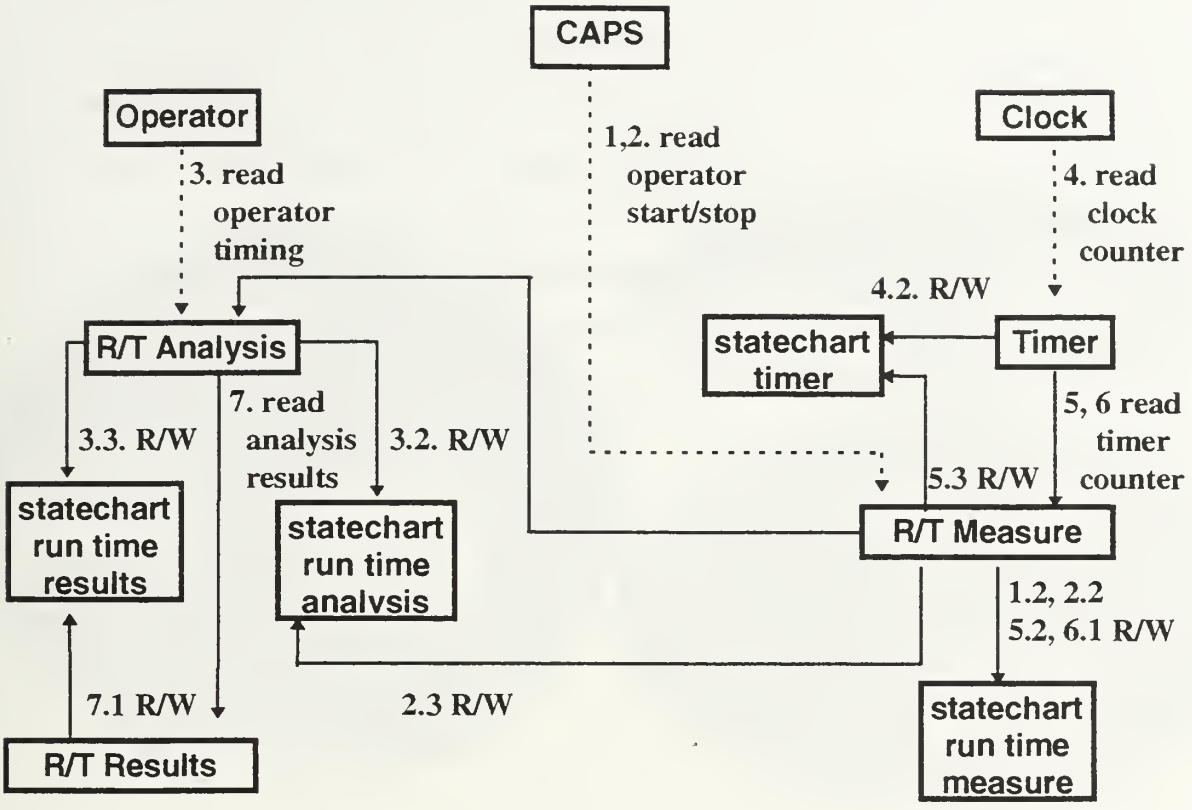

Figure 26. Real-Time Event Execution Monitoring System Event Threads.

The event thread sequence begins with Sequences one and two which represent read operator start and operator stop. This read takes place from the CAPS object to the Run Time Measure object. This exchange allows the Run Time Measure object to obtain precise operator execution times. State changes during these interactions are recorded and read from the statechart run time analysis and the statechart run time measure.

Sequence three represents a read of the Operator object by the Run Time Analysis object. The Operator object contains planned run time data which must be utilized in the calculation of planned Vs actual run time of a specific operator. State changes resulting from these interactions are written to, and read from two state charts. The statechart run time analysis and the statechart run time results are used for sequence three event thread activities.

The Clock object is read by the Timer object in Sequence four. The values of the counter data found in Clock object are utilized by the Timer object to establish start and stop times for the execution of the Operator object. The state chart utilized for state changes resulting from the interaction of these objects is the statechart timer. This state chart is written to by the Timer object and read by the Timer object.

Interaction between the Timer object and the Run Time Measure object is noted as Sequences five and sequence six. The Run Time Measure object reads the Timer object counter value at specific times to mark the start and stop of the Operator object execution. The changes to states are noted in two state charts. The timer activity changes in state are written to statechart timer, which the Timer object also reads from. The state changes in the measurement object called Run Timer Measure are written to and read from the statechart run time measure. The statechart timer is also read by the Run Time Measure object.

F. OBJECT GROUPING

The object grouping for the Real-Time Event Execution Monitoring System is based upon the rules for determining a fair set of object groups found in the Object Oriented Technology for Real-Time Systems [12]. The developed object group diagram is described in Figure 27 Object Groups for the Real-Time Event Execution Monitoring System shown below.

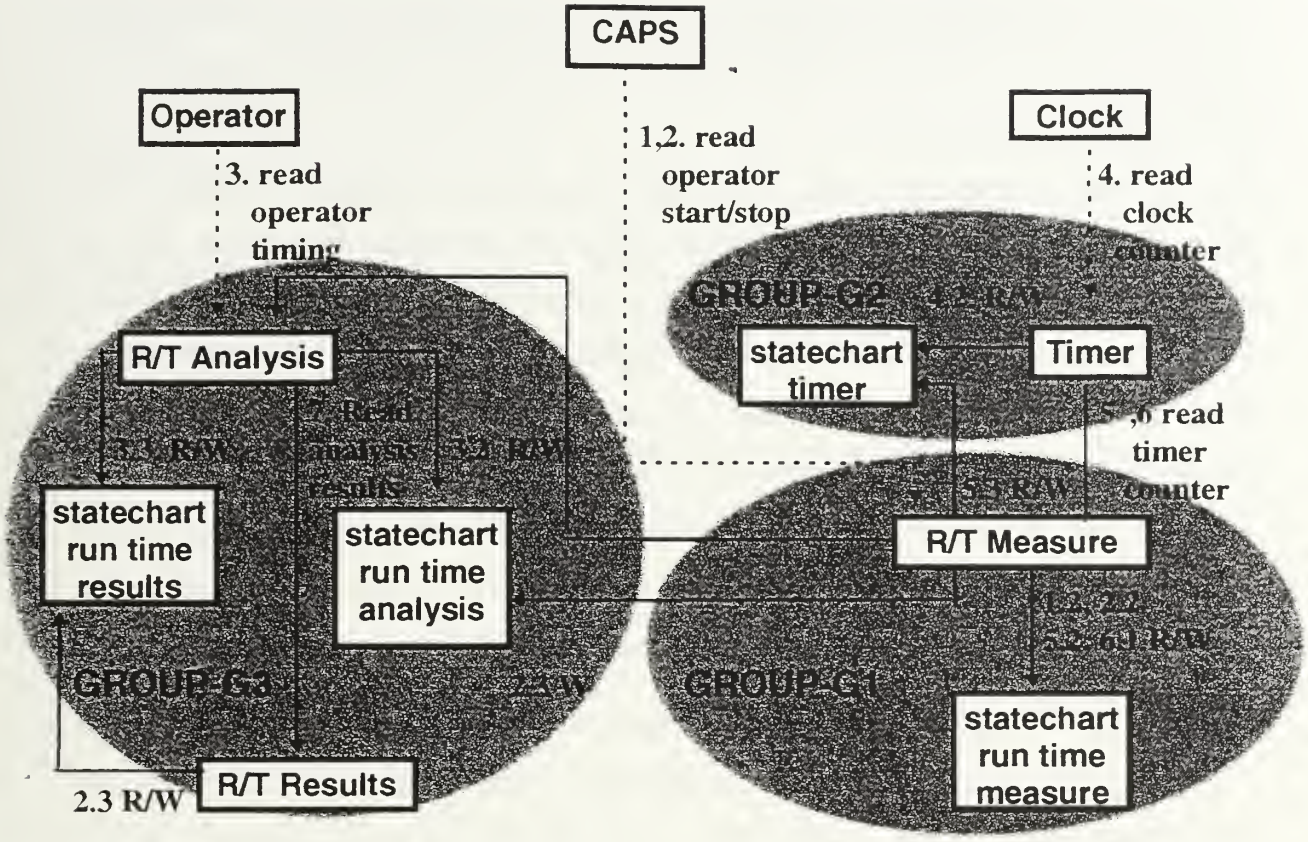

Figure 27. Object Groups for the Real-Time Event Execution Monitoring System.

The group G1 is composed of one object and one state chart. The Run Time Measure object is one of the main objects within the Real-Time Event Execution Monitoring System program. This object is responsible for interacting with the main Timer object, the CAPS object and the Run Time Analysis object. The statechart run time measure serves to hold state changes.

The object grouping listed as the group G2 also contains one object and one state chart. This group G2 is responsible for direct interaction with the system Clock object as well as the Run Time Measure object from the group G1. The statechart timer will hold all state changes for the Timer object. This state chart is a shared object. The statechart timer

is accessed by both the Run Time Measure object from the group G1 and the Timer object from the group G2. This access consists of both read and write on the part of both G1 and G2 group objects.

Group G3 consists of two objects and two state charts. The Run Time Analysis object is also a critical part of the Real-Time Event Execution Monitoring System program. The object directly interacts with the Operator object as well as the Run Time Measure object from G1 and the Run Time Results object within G3. The state charts within this group G3 are the statechart run time analysis and the statechart run time results. The statechart run time analysis is a shared object with group G3 and group G1. This state chart is accessed by both the Run Time Analysis object from the group G3 as well as the Run Time Measure object from the group G1. This access consists of read and write on the part of the Run Time Analysis object in G3 and write only on the part of the Run Time Measure object in G1.

G. OBJECT SHARING

The sharing of objects within the Real-Time Event Execution Monitoring System software is intentionally kept to a minimum. Furthermore the only entities to be shared between object groups are not, in the most strict interpretation, exactly objects themselves. These entities, which are shared among the object groups, are certain State Charts. The *statechart run time analysis* as well as the *statechart timer* are both written to and read from among the different object groups. The required selected grouping of objects and resultant independent execution of these objects from differing paths elicits the sharing of the objects as depicted in the Shared Object Table found inTable 8 below.

Object	Shared Between Groups	Solution
statechart run time analysis	G1 - G3	Uncritical
statechart timer	G1 - G2	Uncritical

Table 8. Real-Time Event Execution Monitoring System Shared Object Table.

The direct interaction between the object group G1 and the object group G3 calls for the statechart run time analysis to be shared. The Run Time Measure object interacts with the statechart run time analysis by writing to it and the Run Time Analysis object performs both writing and reading operations on the state chart. This sharing interaction between objects groups of the statechart run time analysis is not considered critical to synchronization, and no context or interrupt blocking or disabling is recommended for this design.

The direct interaction among the object group G1 and the object group G2 necessitates that the statechart timer be shared. The Timer object interacts with the statechart timer by both writing to the state chart and reading from the state chart. The Run Time Measure object also interacts to the statechart timer by performing both read and write functions on the state chart. As in the case of the statechart run time analysis, the sharing interaction between objects groups of the statechart timer is not considered to be critical to synchronization, and no context or interrupt blocking or disabling is advocated in this design as is shown inTable 8.

H. CODE MAPPING

This section represents the control structure pseudo code for the Real-Time Event Execution Monitoring System program. This pseudo code exemplifies the Code Mapping which is based upon the earlier design segmentation representing the G1, G2, and G3 Object Groupings. The Code Mapping consists of linking each of these groups to an Ada task, such as the CAPS static scheduler. The Code Mapping which will be concluded in the implementation phase of the Real-Time Event Execution Monitoring System program will be based upon Ada objects, functions, and procedures rather than processes as shown in the pseudo code found in code mapping Appendix E. Communications between object groups will be implemented as Ada procedure calls and is represented in the pseudo code as such. The pseudo code found in the Appendix E is used to describe the control structure of each task within its group.

Beginning this pseudo code narration from the perspective of the group G1, the initial sequence of steps is to initialize a real-time timer. This is accomplished via the CREATE_NEW_RT_TIMER call which returns an initialized real-time timer. The new real-time timer is then started and placed in the running state using the START_RT_TIMER call. Prior to getting the execution start time the timer is reset and the counter is set to zero using RESET_RT_TIMER call. All the above calls are from the RTMEASURE object to the TIMER object, group G1 to group G2 respectively. Next the RTMEASURE object sends a get start time command to TIMER object. This is accomplished, again, via communication between the group G1 and the group G2. After the read operator execution start the next step is to read the time counter from the TIMER object to the RTMEASURE object. Checking for invalid operator data is performed and flagged if present. The value for operator start time in the record type RUN_TIME_RECORD.

Continuing on, the next step after the operator has finished executing is to send execution command reading TIMER elapsed time from RTMEASURE object to TIMER object. The communication is between the group G1 and the group G2 as implemented.

The read of the elapsed time counter from the TIMER object to the RTMEASURE object is performed to get the stop time counter data.

The value for operator execution count is incremented and stored along with operator finishing time are stored within the record type RUN_TIME_RECORD. Next an synchronous communication between the group G1 and the group G3 is achieved via the GET_OPERATOR_EXECUTION_DATA call. This enables G1 to send the Operator run time execution data to G3. This transfer is from the RTMEASURE object to the RTANALYSIS object.

The perspective of pseudo code from the group G2 begins with the reception of the call to create a new TIMER from the RTMEASURE object group G1. Upon receiving this call the TIMER object establishes a new record type TIMER_RECORD, adds the timer to the timer list and returns a new timer to the RTMEASURE object. Next the RTMEASURE object calls the TIMER object to start the newly created timer and reset the timer prior to operator execution start. Next the G2 group TIMER object performs a read counter function from clock object to TIMER object and checks for invalid system clock data.

The reception of the command to stop timer execution from the group G1 RTMEASURE object to the group G2 TIMER object is performed. Next the system clock is read from the clock object to the TIMER object and the elapsed time is recorded.

Group G3 is responsible for performing an analysis on the execution time data and provide a run time result to the RTRESULT object. This process is initiated by reading the predefined operator timing data from the operator object and checking for validity. Once this has been accomplished the planned run time data is stored in the record type RUN_TIME_RECORD.

The next step in the analysis process is to retrieve the operator run time data from the RTMEASURE object group G1. This is achieved by using the GET_OPERATOR_EXECUTION_DATA procedure, once received the data is checked for validity and stored in the record type RUN_TIME_RECORD. This record is then forwarded on to the ANALYZE_RESULTS_DATA procedure within the group G3 module. This procedure performs calculations upon the operator run time data to

determine various operator characteristics such as run time averages, maximums, and execution count. This data is also stored within the RUN_TIME_RECORD structure.

The RTANALYSIS object group G3 then transmits the analysis information to RTRESULT object also within group G3 with the SEND_OPERATOR_RUN_TIME CALCULATION_DATA call. The construction of a logfile is undertaken for writing the tabulated operator run time execution data. The logfile is opened and the data within the RUN_TIME_RECORD is written to the file. When finished the logfile is then closed.

This analysis information is formulated into run time execution output for transmission to the user object for run time statistical review for program timing analysis (i.e. adjust scheduler to better fit actual execution run time.)

I. CLASS SPECIFICATION

Within the Real-Time Event Execution Monitoring System project there are four object classes. These classes include: RTANALYSIS; RTMEASURE; RTRESULTS; and TIMER. Ada Language class package specifications have been developed for each of these object classes. These package specifications include definitions of public and private attributes, as well as definitions for functions and procedures of the Real-Time Event Execution Monitoring System object classes. The Ada class package specification for the objects can be seen in Appendix E Class specification

The object RTANALYSIS includes the GET_OPERATOR_EXECUTION_DATA procedure, the procedure called GET_OPERATOR_TIMING_DATA, and the Ada procedure titled SEND_OPERATOR_RUN_TIME_CALCULATION_DATA. This object is contained within the Ada RUN_TIME_ANALYSIS package which is also comprised of an establishment of a record type "RUN_TIME_RECORD", which is defined in the RUN_TIME_MEASURE package, for synchronous intergroup data communication. The communication between the group G1 and the group G3 is accomplished within the

procedure GET_OPERATOR_EXECUTION_DATA by the record type RUN_TIME _RECORD. The procedure ANALYZE_RESULTS_DATA performs an analysis upon the operator run time execution data previously collected. The results of the specific analysis is stored within the respective record type RUN_TIME_RECORD subtype element.

The RTMEASURE object includes the real time timer related procedures CREATE_NEW_RT_TIMER; procedure START_RT_TIMER, and RESET_RT_TIMER which are for the establishment of the real-time timer, the starting of the real-time timer and the elapsed time counter reset of the real-time timer respectively. The Ada procedure GET_OPERATOR_EXECUTION_START, and the Ada procedure GET_OPERATOR_EXECUTION_STOP provide for timing interaction between the CAPS object, the TIMER object and RTMEASURE object. The procedure SEND_OPERATOR_EXECUTION_DATA is used to provide the synchronous communication link between the group G1 and the group G3. Within this procedure the record type for sending data from G1 to G3 is RUN_TIME_RECORD is composed of the subtypes: Operator_Name (representing the current operator under run time execution analysis), Execution_Start/Execution_Stop (representing the operator execution starting and finish times), Planned_Start/Planned_Stop (representing the operator statically scheduled start and finish times), Actual_Run_Time/Planned_Run_Time (representing the operators actual execution time and the operators pre-scheduled execution time), Overhead (representing the current run time monitoring intrusion time), Total_Run_Time (representing the cumulative execution run time of the operator), Total_Timing_Error (representing a running total of operator run time errors including missed deadline), Average_Timing_Error (a running average of the operator timing errors), Average_Run_Time (representing average operator execution run times), Maximum_Run_Time (the largest execution time recorded for the specific operator), Execution_Count(total number of operator firings), and Run_Time_Difference (planned Vs actual operator execution run time). This RUN_TIME_RECORD record is public and is shared by all the Real-Time Event Execution Monitoring System modules. Lastly the procedure SUBTRACT_TIMER allows for the subtraction of time of type "Duration"

from the real-time timer elapsed time element. All of these elements are contained within the Ada package RUN_TIME_MEASURE.

The TIMER object handles all of the creation and administration of real-time timer functions. The TIMER object contains procedures, functions and data types patterned after the CAPS PSDL_TIMERS module[†]. The major difference between these two modules is that of the Ada package used by the respective module. The PSDL_TIMERS module is based upon the ADA.CALENDAR package and the RUN_TIME_TIMER module is based upon the ADA.REAL_TIME package. The TIMER object utilizes structure consisting of record type TIMER_RECORD. The subtype are START_TIME of type ADA.REAL_TIME.Time, ELAPSED_TIME of type ADA.REAL_TIME.DURATION, and PRESENT_STATE which is an enumerated type. For real-time timer handling the TIMER object utilizes the NEW_TIMER function to create and initialize a new real-time timer entity, procedure START to start the timer counter, procedure RESET to zero the timer counter, and the function HOST_DURATION to read the total time accumulated in the real-time timer on the host machine. The synchronous communication between the group G2 and the group G1, as represented by the above objects, is accomplished by record type RUN_TIME_RECORD and the procedure call HOST_DURATION in combination with the RTMEASURE object procedures GET_OPERATOR_EXECUTION_START, GET_OPERATOR_EXECUTION_STOP. The above elements are all contained within the RUN_TIME_TIMER Ada package.

The simplest of the entire group of Ada language class package specifications for the Real-Time Event Execution Monitoring System contains six procedures. The object class called RTRESULTS is specified within the Ada package RUN_TIME_RESULTS. Within this object are the procedures which manipulate the Real-Time Event Execution Monitoring System logfile. The procedure Build_Log_File initially creates a logfile for storage of the Operator execution monitoring and analysis results. The Open_Log_File procedure performs an open call on the previously created logfile at periodic intervals. Once open, the logfile is populated with Operator execution run time data via the

[†] See Chapter V, Section A, Design Decisions

procedure Write_Log_File, the file is then closed by utilization of the Close_Log_File procedure. The procedure GET_OPERATOR_RUN_TIME_CALCULATION_DATA is utilized for intergroup communication with the RTANALYSIS object, the record type RUN_TIME_RECORD is used to pass data between the RTANALYSIS object and the RTRESULTS object. The GET_OPERATOR_RUN_TIME_CALCULATION_DATA procedure, together with the log file handling procedures formulate the RTRESULTS object and are contained within the RUN_TIME_RESULTS Ada package. The RTANALYSIS and RTRESULTS together perform the tasks of receiving and calculating the operator run time and transmitting the results to the User for analysis.

V. IMPLEMENTATION CONSEDERATIONS

The implementation of the Real-Time Event Execution Monitoring System source code utilized the ADA programming language. Initial work for this project source code software development was undertaken by using the Ada83 libraries. This source code development effort was later completed by reworking the system based upon the Ada95 libraries with real-time annex constructs.

To monitor the run time execution of real-time task sets the Real-Time Event Execution Monitoring System program leveraged off previously mentioned related work by the strategic placement of profiling points within the CAPS static scheduler "STATIC_SCHEDULE" task. However, this approach raises the question of introducing an additional set of overhead intrusion into the system. Issues such as the competition for real-time CPU resources, and additional context switching must be acknowledged. The monitoring intrusion resulting from implementation of the Real-Time Event Execution Monitoring System program is discussed further in chapter VI conclusion & future research.

A. DESIGN DECISIONS

This section includes project design decisions and modifications which are related to the Real-Time Event Execution Monitoring System project development and design effort.

The initial project decision was made to utilize the Object Oriented approach called OCTOPUS for the development of the Real-Time Event Execution Monitoring System project. The decision was based upon a choice between the CAPS/PSDL approach and the OCTOPUS approach. The chosen OCTOPUS method is a hybrid of OMT and Fusion methods, and intended for real-time design.

The requirement analysis of run time execution monitoring work was initiated by deciding to pursue the approach of isolating and determining the system goals for run time

execution monitoring system. Subsequently it was decided to rework the goals to broaden the systems objectives for the Real-Time Event Execution Monitoring System project to include more refined descriptions. These revised project goals will allow for the transmission of additional timing information to the system designer for utilization in the CAPS users prototype design effort. Also included additional sub-goals into the goals section.

The approach to the system architecture followed a partition method. The partitions represented modular components created to separate the tasks of analyzing the Real-Time Event Execution Monitoring System. The Real-Time Event Execution Monitoring System was accordingly decomposed into various independent system domains. This decomposition resulted in the following independent system domains: User Interface Domain, Device Domain, Measurement Domain, and the Analysis Domain. To allow for loose module coupling the communication between these previously segmented domains the utilization of a formal parameter passing method was followed.

For the Object Model construction an iterative process for the development of the Object Model was utilized. This approach followed a method where the System Use Cases were first examined, scanning for nouns to represent system objects. These objects were then formulated from this first pass into a list. The resulting object list was then compared to the previously developed Use Case objectives to determining the relevancy of each object. In this way the object list was then prepared to be completed.

The development of the functional model utilized the previously created operation sheets. This allowed for the Real-Time Event Execution Monitoring System to present a more accurate description of functional interface of the subsystems. The services provided by a given subsystem of the Real-Time Event Execution Monitoring System are depicted within this interface. The interfaces spans across the subsystem application boundary, between other subsystems and external agents of the Real-Time Event Execution Monitoring System.

In a similar way the development of the dynamic model for the Real-Time Event Execution Monitoring System was based upon the initial analysis of events through the

creation of event lists and event sheets. This was followed by utilization of state charts to provide event analysis, and validated the dynamic model by use of scenarios.

The initial task for describing object interaction was accomplished by the construction of Event Threads. Developed the Event Threads for the Real-Time Event Execution Monitoring System by following the recommended step-by-step approach for building event threads when using the OCTOPUS method. The resulting event threads which were created included: ReadOperatorStart; ReadOperatorStop; ReadOperatorData; GetClockTime; ReadTimerStart; and ReadTimerStop.

This was followed by the development of object interaction graphs which were based upon each of the previously created event threads within the Real-Time Event Execution Monitoring System. The object interaction graphs were also constructed using the lists of events which were previously developed within the System Requirements document. The Object Interaction Graphs were produced to illustrate the detailed representation of the Real-Time Event Execution Monitoring System object interaction.

The development of the object grouping for the Real-Time Event Execution Monitoring System was based upon the rules for determining a fair set of object groups found in [12]. The control structure pseudo code for the Code Mapping was based upon the earlier design segmentation representing the G1, G2, and G3 Object Groupings. The Code Mapping consisted of linking each of these groups to an Ada task. The communications between object groups will be implemented as formal parameter passing based upon a record type RUN_TIME_RECORD.

For the development of the Real-Time Event Execution Monitoring System program source code software is was initially decided to utilize the CAPS environment to develop a prototype version. This was to be accomplished by transposing tasks into operators and communication threads into streams. Eventually this approach was considered to be not feasible due to the state of version upgrade which the CAPS environment was undergoing. Summarily the utilization of the CAPS tool to develop a prototype was abandoned.

In place of utilizing the CAPS environment for source code development the was accomplished within a simple UNIX environment with the basic compiler and standard

debugging tools. The language of choice for this code development and implementation work was Ada. The Ada83 libraries were used for the initial project source code software development. This development effort was later changed to utilization of the Ada95 libraries with real-time annex constructs. The final implementation was created based upon these Ada95 libraries to provide a finer granularity in clock ticks.

Additionally it was determined that the most efficient method for implementation of the REAL_TIME_TIMER package would be to leverage off the existing PSDL_TIMERS package. The rational for this decision was twofold. Since the PSDL_TIMERS package was implemented by utilizing the ADA.CALANDER libraries the future utilization for CAPS environmental development which required using real-time constructs found in the ADA.REAL_TIME libraries would be less cumbersome as the REAL_TIME_TIMER package was based upon the latter. Additionally, the PSDL_TIMERS package was written soundly and would provide a good base for the timing measurement effort.

VI. CONCLUSION & FUTURE RESEARCH

This thesis successfully introduced a run time execution monitoring program into the CAPS embedded real-time software development environment component called the static scheduler tool. This enhancement to the CAPS static scheduler enabled a significantly improved feedback of real-time event execution data to the system user. The task run time execution response data included: timing information on task execution start and finish; underallocation/overallocation of statically scheduled task execution times; total number of timing errors during execution; run time monitor intrusion overhead resource utilization; total number of operator firings; and average/slowest/fastest run time execution data.

The Real-Time Event Monitor program has been successfully applied to a specific targeted prototype software developed under the CAPS environment. This has been accomplished by instrumenting the CAPS static scheduler module with profiling points which transmit all run time execution data to the various modules within the Real-Time Event Execution Monitoring System program. All resulting run time data has been obtained from the operation of the targeted prototype software.

The targeted prototype software has been "fine tuned" by transmission and analysis of the run time execution data provided by the Real-Time Event Execution Monitoring System. This fine tuning consisted of an iterative process, as shown in Figure 28 . This process isolated an appropriate time segment for the targeted prototype software Operator execution. Once the optimal execution time segment has been determined it is then verified by prototype software operation.

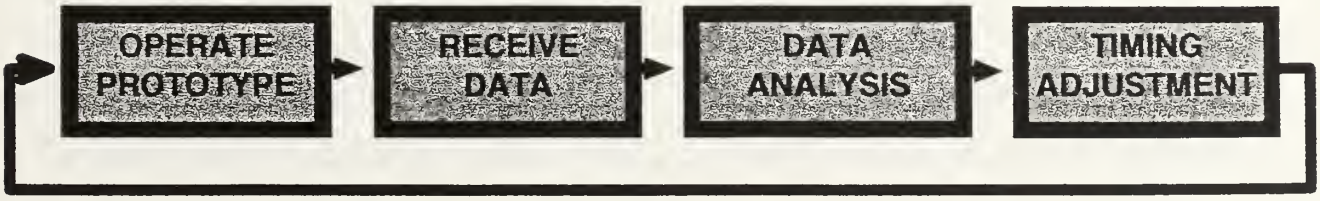

Figure 28. Run-Time Execution Data Isolation.

The successful employment of the Real-Time Event Execution Monitoring System required that a careful examination be performed upon the resulting intrusion behavior and overhead resource utilization. The actual collection of task run time execution data required intrusion of two simple procedure calls. The remaining analysis and tabulation of the run time execution data required a much more burdensome imposition. Initially the overhead intrusion was significant enough to introduce timing errors into the static scheduling module. However, this difficulty was overcome through the implementation of an intrusion minimization method. Following this approach required the active synchronization of the real-time timer and the scheduler timers. The data collection overhead figures range from 301 microseconds to 347 microseconds. The run time execution data analysis and tabulation overhead figures range from 2.215 milliseconds to 1.834 milliseconds. At each monitoring intrusion point the scheduler timer was stopped and summarily restarted once the monitoring activity was completed. Additionally the majority of the overhead processing costs were greatly minimized by placement of the analysis and tabulation tasking to occur at the end of the harmonic block segment, just prior to the scheduling timer reset. Through the implementation of the intrusion minimization method the computational overhead required for the operation of the Real-Time Event Execution Monitoring System was determined to be not excessive.

This thesis demonstrates that run-time statistics can successfully be collected during the execution of a real-time prototype without imposition of an excessive computational overhead. There were however some initial complications in the actual extraction of overhead intrusion data figures as is mentioned in later paragraphs within this section.

This thesis has also determined that real-time event monitoring is an effective tool for improving the maximum execution time assignment for real-time task sets. To illustrate this an experiment has been performed upon real-time prototype software. The software for this study is a real-time temperature controller. The temperature controller is a simplified program comprised of two real-time operators. The program operation consists of a sensor operator which continually monitors temperature changes and an evaluation operator that performs an analysis on the sensor output. The heater or cooler

mechanisms are adjusted by signals from the eval operator. Illustrated below in Figure 29 is an elementary dataflow diagram describing the temperature controller prototype program. The Sensor operator and the Eval operator are time-ctirical, each has a period of 1 millisecond. The three numbers listed beneath the Sensor operator and Eval operator respectivly represent three seperate sets of maxmium execution times. The operator's execution was statically scheduled.

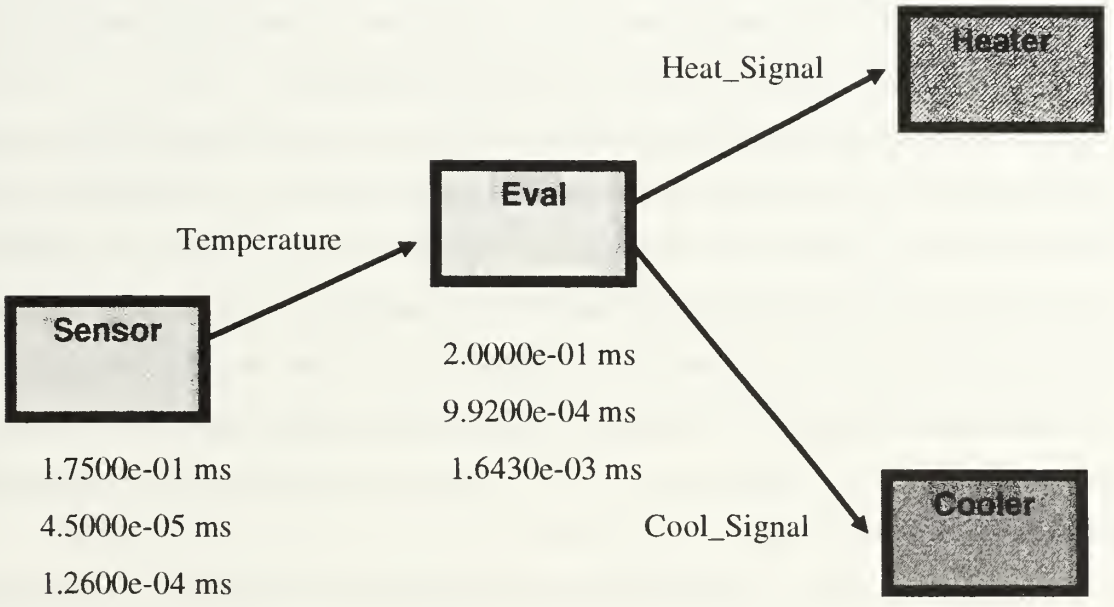

Figure 29 Prototype Program Experiment

A series of three separate test experiments have been performed upon the temperature controller prototype. The results can be observed in the Operator Run Time Analysis Logfile listed in Appendix H. This logfile represents the output for the three separate prototype program execution timing adjustment iterations in which the maximum execution time constraint has been altered without modification to the period.

During the 1ST RUN the operator Sensor planned execution time was set to 175.0 milliseconds. This initial planned time proved to be far in excess of even the 320.0 microsecond maximum run time recorded. The figures for the minimum execution time of 8.8 microseconds and an average execution time of 120.0 microseconds were also recorded during this run of the Sensor operator. The 1ST RUN data for the operator Eval also proved that the planned execution time of 200.0 milliseconds was excessive. A maximum execution time of 2.078 milliseconds, a minimum execution time of 1.199 milliseconds, and an average execution time of 1.3756 milliseconds were recorded for this run. Additionally both operators incurred no timing errors while their execution time included the excessive slack afforded by the initial planned times.

Based upon this feedback provided by the first run, the maximum execution time of the Sensor operator and the Eval operator were changed to 45.0 microseconds and 90.0 microseconds respectivly. However, the resulting run time included a significant increase in the number of timing errors incurred. The maximum, minimum and average run time for the Sensor operator were 260.0 microseconds, 90.0 microseconds, and 121.667 microseconds respectively. The maximum, minimum, and average run time for the Eval operator were 1.56 milliseconds, 1.137 milliseconds, and 1.22183 milliseconds respectively.

Again utilizing the Operator Run Time Analysis Logfile as a feedback mechanism, an observation of the data obtained from the second run clearly shows that the planned execution times for the sensor and eval operators have been adjusted too low. Timing errors are occurring at almost every execution period. Accordingly the 3RD RUN of the Sensor operator with the planned execution time set more closely to the actual observed run times, has provided a more likely fit. The planned execution time of 126.0 microseconds worked well without a significant increase in the occurrence of timing errors. The maximum, minimum, and average run times were 131 microseconds, 91 microseconds, and 96.8 microseconds respectively. For the 3RD RUN of the Eval operator the planned execution time was also changed to 1.643 milliseconds. This change resulted in no significant increase in the occurrence of timing errors as compared to the

first run. The maximum, minimum, and average run times were 1.511 milliseconds, 1.221 milliseconds, and 1.2876 milliseconds respectively.

This thesis has demonstrated the viability of applying the Ada95 libraries and the Monotonic Time real-time extension annex for critical close tolerance measurement requiring fine time within microsecond level granularity.

The variation in run time for specifically isolated task set created within the CAPS environment has also been investigated. This variance within run time execution of the CAPS operator ranges from 23 to 29 microseconds. This thesis concludes that this real-time task run time execution variation is due to conditional factors outside of the environment which the prototype development is being performed. The platform CPU being a shared resource, hence the CAPS program does not have exclusive utilization rights. Therefore the CAPS (Real-Time Event Execution Monitoring System, Static Scheduler, etc) environment is essentially swapped out of the CPU periodically being replaced by other local processes as local interrupts are periodically generated. Thus, the variance is not large enough to be significant for purposes of prototype development within the CAPS environment. If finer grain development work must be accomplished a dedicated system is recommended.

Many topics still remain and future work in this area should address these areas. There is a need to upgrade the entire grouping of PSDL timer modules within the CAPS environment from the ADA.CALENDAR timing package to the ADA.REAL_TIME timing package. This task will allow CAPS to become more aligned with the Real-Time Event Execution Monitoring System program real-time approach. This work will provide for an improved timing granularity and increase the eventual prototype development accuracy.

There is also a need for an enhancement of the task run time execution feedback mechanism. The specific task data information should be presented in a separate "ANALYSIS" window within the CAPS environment to allow easy user access.

The accurate determination of overhead intrusion into the static scheduler for run time execution monitoring is currently very cumbersome. The immediate program must be run from within the static scheduler and separate from the prototyping process to

determine the overhead intrusion. The resulting data is not without inaccuracies and allows some variation. Future work in this area should address a dynamic collection of intrusion data figures. The record type RUN_TIME_RECORD which allows for storage of operator execution data has been altered to include the subtype labeled Overhead. Any future effort to dynamically collect intrusion should make use of this structure to allow a more accurate determination of overhead calculation.

The Real-Time Event Execution Monitoring System program should be fully integrated into the CAPS environment. This work will include establishing a linkage with the PSDL editor.

APPENDIX

A. USE CASE SHEETS

==========

Use Case	Schedule R/T task
Actors	CAPS R/T SCHEDULERS
Precondition	Real-Time Operator created
Description	Schedule real-time task, check for valid MET, MRT, PERIOD Exception if invalid schedule. Check for parent task.
Subuse cases	
Exceptions	Produce error message
Activities	
Postcondition	Task given a valid real-time schedule, placed in execution queue if successfully completed.

==========

Use Case	(U1)Activate Operator Run Time Program
Actors	System Designer, User/Designer
Precondition	Real-Time Operator previously created
Description	Start operator run time measurement program. Exception when no designated Real-Time Operator can be found.
Subuse cases	
Exceptions	System responds to Exception by producing an error message, do not proceed with timer activation.
Activities	
Postcondition	Operator run time measurement program started, prepare to activate timers upon successful completion. Unsuccessful completion produce error message

==========

Use Case	(U2)Start Measurement Timer
Actors	System Clock
Precondition	Valid Real-Time Operator previously created, and Operator Run Time Program started.
Description	Start clock timing when real-time operator starts its execution. This task includes that of providing accurate real-time timing information. The timing requirements are to provide granularity at least fine enough to allow accurate run time measurement to be performed. Timing requirements also include task to execute at same

time as operator starts executing. Exception when task fails to execute concurrently with operator.

Subuse cases

Exceptions System responds to Exception by producing an error message, do not proceed with run time measurement.

Activities

Postcondition Activated timer to capture operator run time data upon successful completion. Unsuccessful completion produce error message

==========

Use Case (U3)Stop Measurement Timer

Actors System Clock

Precondition Valid Real-Time Operator previously created, Operator Run Time Program started, and Start Measurement Timer started and successful.

Description Stop clock timing when real-time operator completes its execution. This task includes that of providing accurate real-time timing information. The timing requirements are to provide granularity at least fine enough to allow accurate run time measurement to be performed. Timing requirements also include task to execute at same time as operator stops executing. Exception when task fails to execute and stop timer.

Subuse cases

Exceptions System responds to Exception by producing an error message, proceed with Reset Measurement Timer task activation.

Activities

Postcondition Timer stopped, proceed with capture of run time data for analysis, and Reset Measurement Timer task upon successful completion. Unsuccessful completion produce error message

==========

Use Case (U4)Reset Measurement Timer

Actors Autonomous activity of system

Precondition Start Measurement Timer started, and Real-Time measurement timer task previously stopped successfully

Description Reset timer when run time measurement completes its execution. The task is part of providing accurate real-time timing information. Timing requirements include this task to execute before next operator starts executing. Exception when task fails to reset the timer.

Subuse cases

Exceptions System responds to Exception by producing an error message, do not proceed with Measurement Timer task activation.

Activities

Postcondition Timer reset, proceed with start of run time measurement upon successful completion. Unsuccessful completion produce error message

==========

Use Case	(U5)Get Operator Planned Run Time Characteristics
Actors	System Designer, R/T Scheduler, Operator
Precondition	Valid Real-Time Operator created and R/T characteristics prespecified
Description	Retrieve the planned run time characteristics of real-time operator check for timing constraint properties MET, MRT, PERIOD, FW, MCP, MOP Exception if invalid characteristics.
Subuse cases	
Exceptions	System responds to Exception by producing an error message to user, do not proceed with Operator run time analysis task activation.
Activities	
Postcondition	Real-time operator characteristics can now be compared with actual run time results. Unsuccessful completion produce error message

==========

Use Case	(U6)Analyze Operator Run Time
Actors	Autonomous activity of system
Precondition	Planned Real-Time Operator data previously collected, and Operator Run Time data compiled.
Description	After collecting actual and planned Operator run time data an analytical comparison will be performed between actual and planned execution times. Exception will occur when comparison invalid
Subuse cases	
Exceptions	System responds to Exception by producing an error message, do not proceed with transmission of data to user task activation.
Activities	
Postcondition	Comparison data will have been calculated based upon actual Operator run time Vs planned. Unsuccessful completion produce error message

==========

Use Case	(U7)Transmit Operator Run Time data to User
Actors	System Designer, User/Designer
Precondition	Real-Time Operator data previously correctly collected and analyzed.
Description	At the completion of Operator Run Time data collection, and the finish of Operator Execution, the actual and planned run time analysis data for given operator will be presented. Exception will occur when comparison invalid.
Subuse cases	
Exceptions	System responds to Exception by producing an error message, operator run time data transmission to user replaced by error code.
Activities	
Postcondition	Display run time data analysis for user viewing upon successful completion. Unsuccessful completion produce error message

==========

B. OPERATION SHEETS

Operation	(O1) CalculateOperatorRunTime
Description	Using the timer data which measured operator start and stop times the actual run time of the operator is subsequently calculated.
Associations	Object R/T Measure, Operator, R/T Results, R/T Analysis
Preconditions	Accurate measurement of operator start/stop execution times must have been completed. Operator planned timing data must have been developed.
Inputs	Operator execution start and execution completion time, Operator planned timing start and completion times.
Modifies	No modification to the operations argument, no modification to common data within the subsystem.
Outputs	Operation result -Operator Run Time forwarded to R/T Results Object
Postcondition	If valid operator run time timer data is obtained and planned run run time is read then the calculation of the difference (if any) can proceed. Upon error condition no calculation takes place.

Operation	(O2) SetStartTime
Description	Start timing clock for operator run time execution measurement
Associations	Object Clock, R/T Measure, Timer
Preconditions	Accurate system clock with fine granularity.
Inputs	Signal from R/T Measure
Modifies	No modification to the operations argument, modification to common data within the subsystem includes update time counter
Outputs	Start time forwarded to R/T Measure Object
Postcondition	If accurate clock resolution is available, start timer is set. Upon error condition no timer start takes place.

Operation	(O3) SetStopTime
Description	Stop the timing clock for operator run time execution measurement
Associations	Object Clock, R/T Measure, Timer
Preconditions	Accurate system clock with fine granularity.
Inputs	Signal from R/T Measure
Modifies	No modification to the operations argument, modification to common data within the subsystem includes update time counter
Outputs	Clock Stop time forwarded to R/T Measure Object
Postconditions	If accurate clock resolution is available, stop timer is set. Upon error condition no timer stop (or start).

Operation	(O4) ResetTime

Description Reset the timing clock for operator run time execution measurement
Associations Object Clock, R/T Measure, Timer
Preconditions Accurate system clock with fine granularity.
Inputs Signal from R/T Measure
Modifies No modification to the operations argument, modification to common data within the subsystem includes zeroed time counter
Outputs Successful reset of timer code forwarded to R/T Measure Object
Postconditions Assuming accurate clock resolution, a reset of the timer takes place. Upon error condition no reset of timer -Real-Time Event Monitoring System notified with error code.

==========

Operation (O5) ReadOperatorData
Description Get the real-time operator planned execution time.
Associations Object Operator, R/T Measure, R/T Results, R/T Analysis
Preconditions Valid Operator real-time characteristics execution times must have been completed.
Inputs Real-time operator planned execution run time.
Modifies No modification to the operations argument, no modification to common data within the subsystem.
Outputs Operation result -Operator planned Run Time forwarded to R/T Analysis Object
Postconditions If valid operator exists, then read of operator execution time data takes place. Upon error condition no read of data. -Run Time Monitoring System notified with error code.

==========

Operation (O6) SendOperatorExecutionData
Description Transmit the real-time operator execution analysis.
Associations Object, R/T Results, R/T Analysis, System Designer
Preconditions Valid real-time Operator run time execution data must have been analyzed and sent.
Inputs Real-time operator execution run time data.
Modifies No modification to the operations argument, no modification to common data within the subsystem.
Outputs Operation result -analyzed Operator Run Time transmitted to System Designer Object
Postconditions Upon successful completion of run time analysis and valid calculation of run time, this data is transmitted to System Designer Upon error condition generated error code message is sent.

==========

C. EVENT SHEETS

Event	(E1) ReadOperatorStart
Response	Receive confirming signal that Real-Time operator has started execution. Synchronize the operator start with the timer start.
Associations	CAPS, R/T Measure
Source	R/T Measure
Contents	Timing info for timer synch.
Response Time	Max. 1 ms
Rate	Periodic or Sporadic

Event	(E2) ReadOperatorStop
Response	Receive confirming signal that Real-Time operator has stopped execution. Synchronize the operator start with the timer stop.
Associations	CAPS, R/T Measure
Source	R/T Measure
Contents	Timing info for timer synch.
Response Time	Max. 1 ms
Rate	Periodic or Sporadic

Event	(E3) ReadOperatorData
Response	Receive confirming signal that Real-Time operator has stopped execution. Synchronize the operator start with the timer stop.
Associations	Operator, R/T Analysis
Source	R/T Analysis
Contents	Analysis information data on planned operator execution run time.
Response Time	Max. 1 ms
Rate	Periodic or Sporadic

Event	(E4) GetClockTime
Response	Receive confirming signal that Real-Time operator has stopped execution. Synchronize the operator start with the timer stop.
Associations	Operator, R/T Analysis
Source	R/T Analysis
Contents	Analysis information data on planned operator execution run time.
Response Time	Max. 1 ms

Rate	Periodic or Sporadic
Event	(E5) ReadTimerStart
Response	Receive confirming signal that Real-Time operator has stopped execution. Synchronize the operator start with the timer stop.
Associations	Operator, R/T Analysis
Source	R/T Analysis
Contents	Analysis information data on planned operator execution run time.
Response Time	Max. 1 ms
Rate	Periodic or Sporadic
Event	(E6) ReadTimerStop
Response	Receive confirming signal that Real-Time operator has stopped execution. Synchronize the operator start with the timer stop.
Associations	Operator, R/T Analysis
Source	R/T Analysis
Contents	Analysis information data on planned operator execution run time.
Response Time	Max. 1 ms
Rate	Periodic or Sporadic

D. CODE MAPPING

Group G1:

```
--Initialization of a new real-time timer for execution monitoring
CREATE_NEW_RT_TIMER;

    --Call the Timer object GROUP-G2
     RT_timer := RUN_TIME_TIMER.NEW_TIMER;


--Start the recently created real-time timer
START_RT_TIMER;

   --Call the Timer object GROUP-G2 to start
   RUN_TIME_TIMER.START(RT_timer);

--Zero the real-time timer counter prior to measure start
RESET_RT_TIMER

-- Call Timer object GROUP-G2 for counter reset
   RUN_TIME_TIMER.RESET(RT_timer);


--Retrieve the actual execution starting time of the CAPS operator.
GET_OPERATOR_EXECUTION_START

    -- Initialize start time
    Operator_Data.Execution_Start := 0.0;

    -- Get the real-time timer counter
    Operator_Data.Execution_Start :=
       float(RUN_TIME_TIMER.HOST_DURATION(RT_timer));

   -- Update the execution count
    Operator_Data.Execution_Count := Operator_Data.Execution_Count + 1.0;
```

```
--As the CAPS Operator finishes execution retrieve the actual execution
--finish time of the CAPS operator.
GET_OPERATOR_EXECUTION_STOP

    -- Initialize stop time
    Operator_Data.Execution_Stop  := 0.0;

    -- Get the real-time timer counter
    Operator_Data.Execution_Stop  :=
      float(RUN_TIME_TIMER.HOST_DURATION(RT_timer));

   -- Send execution command start timer execution from RTMEASURE
   -- object to RTANALYSIS object GROUP-G3
  GET_OPERATOR_EXECUTION_DATA(Operator_Data);
```

Group G2:

```
-- Create and initializes new timer upon call from RTMEASURE object GROUP-G1
NEW_TIMER

   --Create new real-time timer record
   result := new TIMER_RECORD;

   --Include the new timer record in the list of timers
   add(result, timers);

  --Return the new timer to the RTMEASURE object GROUP-G1
   return result;


--Begin running the timer counter upon call from RTMEASURE object GROUP-G1
START

     --Set the real-time timer record state and establish start time
     NAME.PRESENT_STATE:= RUNNING;
     NAME.START_TIME:= CLOCK;

--Reset the clock counter to zero upon call from RTMEASURE object GROUP-G1
RESET

     --Set the real-time timer counter to zero and restablish start time
     NAME.ELAPSED_TIME:= 0.0;
     NAME.START_TIME := CLOCK;
```

```
-- Receive the command from the RTMEASURE object to get the execution time
start
--return the elapsed time for start time establishment
HOST_DURATION

    --Set the return value in type duration for RTMEASURE object storage
    NAME.ELAPSED_TIME + CLOCK - NAME.START_TIME;


-- Receive the command from the RTMEASURE object to get the execution time
start
--return the elapsed time for start time establishment
HOST_DURATION

    --Set the return value in type duration for RTMEASURE object storage
    NAME.ELAPSED_TIME + CLOCK - NAME.START_TIME;
```

Group G3:

```
-- Read predefined operator timing data from operator object
GET_OPERATOR_TIMING_DATA

     -- Initialize data variable
     Operator_Data.Planned_Run_Time := 0.0;

     -- Check for valid operator data
     if (Operator_Data.Planned_Stop > Operator_Data.Planned_Start) then

    -- Calculate Planned Run Time
    Operator_Data.Planned_Run_Time :=
            (Operator_Data.Planned_Stop - Operator_Data.Planned_Start);
   else
   -- Put error message
    PUT_LINE("RUN_TIME_ANALYSIS: GET_OPERATOR_TIMING_DATA:
                STOP < START");
   end if;


-- Receive operator execution data from the RTMEASURE object GROUP-G1
GET_OPERATOR_EXECUTION_DATA;
```

```
   -- Initialize data variable
   Operator_Data.Actual_Run_Time := 0.0;

  -- Validity check
  if (Operator_Data.Execution_Stop > Operator_Data.Execution_Start) then

  -- Calculate Actual Run Time
     Operator_Data.Actual_Run_Time :=
                  Operator_Data.Execution_Stop - Operator_Data.Execution_Start;
  else
  -- Put error message
    PUT_LINE("RUN_TIME_ANALYSIS:
GET_OPERATOR_EXECUTION_DATA:
                   STOP < START");
  end if;

  -- Send results to be analyzed from within the RTANALYSIS object
  ANALYZE_RESULTS_DATA(Operator_Data);

-- Perform an analysis upon the recently retrived CAPS operator
-- run time data.
ANALYZE_RESULTS_DATA;

   -- Get total run time
   Operator_Data.Total_Run_Time :=
   Operator_Data.Actual_Run_Time + Operator_Data.Total_Run_Time;

 -- Determine average run time
 Operator_Data.Average_Run_Time :=
     (Operator_Data.Total_Run_Time / Operator_Data.Execution_Count);

 -- Check for new maximum run time
 if (Operator_Data.Actual_Run_Time > Operator_Data.Maximum_Run_Time)
then
   Operator_Data.Maximum_Run_Time := Operator_Data.Actual_Run_Time;
 end if;

 -- Find planned/actual run time difference
 if (Operator_Data.Actual_Run_Time > Operator_Data.Planned_Run_Time) then
   Operator_Data.Run_Time_Difference :=
     Operator_Data.Actual_Run_Time - Operator_Data.Planned_Run_Time;
 else
   Operator_Data.Run_Time_Difference :=
     Operator_Data.Planned_Run_Time - Operator_Data.Actual_Run_Time;
```

```
end if;

 -- Move execution data from RTANALYSIS object to RTRESULTS object
 -- within GROUP-G3
SEND_OPERATOR_RUN_TIME_CALCULATION_DATA;

-- Receive data from analysis procedure
SEND_OPERATOR_RUN_TIME_CALCULATION_DATA;

-- Send the execution data to RTRESULTS object
GET_OPERATOR_RUN_TIME_CALCULATION_DATA;

-- The resultant run time statistics are displayed to the users
SEND_EXECUTE_RUN_TIME_DATA;

    -- Open previously created logfile for writing
    Open_Log_File;

    -- Write the operator execution data to open logfile
    Write_Log_File;
    -- Close logfile when finished writing
   Close_Log_File;
```

E. CLASS SPECIFICATION

```
package RUN_TIME_MEASURE is

 -- Record of operator data
 -- All of the record elements are per instance of each operator.
 type RUN_TIME_RECORD is
  record
    Operator_Name       : String(1..6);
    Execution_Start     : float := 0.0;
    Execution_Stop      : float := 0.0;
    Planned_Start       : float := 0.0;
    Planned_Stop        : float := 0.0;
    Actual_Run_Time     : float := 0.0;
    Planned_Run_Time    : float := 0.0;
    Overhead            : float := 0.0;
    Total_Run_Time      : float := 0.0;
    Total_Timing_Error  : float := 0.0;
    Average_Timing_Error : float := 0.0;
    Average_Run_Time    : float := 0.0;
    Maximum_Run_Time    : float := 0.0;
    Execution_Count     : float := 0.0;
    Run_Time_Difference : float := 0.0;
  end record;
 type RUN_TIME_LIST is access RUN_TIME_RECORD;

 procedure CREATE_NEW_RT_TIMER;

 procedure START_RT_TIMER;

 procedure RESET_RT_TIMER;

 procedure GET_OPERATOR_EXECUTION_START(
                              Operator_Data: in out RUN_TIME_RECORD);

 procedure GET_OPERATOR_EXECUTION_STOP(
                              Operator_Data: in out RUN_TIME_RECORD);

 procedure SUBTRACT_TIMER(sub_time : in Duration);

 procedure SEND_OPERATOR_EXECUTION_DATA(
                              Operator_Data: in out RUN_TIME_RECORD);

end RUN_TIME_MEASURE;
```

```
package RUN_TIME_ANALYSIS is

 procedure GET_OPERATOR_EXECUTION_DATA(
                              Operator_Data : in out RUN_TIME_RECORD);

 procedure GET_OPERATOR_TIMING_DATA(
                              Operator_Data : in out RUN_TIME_RECORD);

 procedure ANALYZE_RESULTS_DATA(
                              Operator_Data : in out RUN_TIME_RECORD);

 procedure SEND_OPERATOR_RUN_TIME_CALCULATION_DATA(
                              Operator_Data:in out RUN_TIME_RECORD);

end RUN_TIME_ANALYSIS;


package RUN_TIME_TIMER is

subtype MILLISEC is NATURAL;
type TIMER is private;
type timer_list is private;

 empty_timer_list: constant timer_list;
 function first(tl: timer_list) return TIMER; -- raises no_subcomponents
 function rest(tl: timer_list) return timer_list; -- raises no_subcomponents
 procedure add(t: TIMER; tl: in out timer_list);

 no_subcomponents: exception;

 function READ(NAME: in TIMER) return MILLISEC;
  -- Timer reading wrt the target machine.
 procedure RESET(NAME: TIMER);
 procedure START(NAME: TIMER);
 procedure STOP(NAME: TIMER);

 -- operations used by the CAPS tools.
 function NEW_TIMER return TIMER; -- creates and initializes a new timer
 function HOST_DURATION(NAME: TIMER) return duration;
   -- total time accumulated in the Timer on the host machine.

 function TARGET_TO_HOST(d: DURATION) return DURATION;
```

```
   -- Converts durations on the target machine to the
   -- corresponding durations on the CAPS host machine.
  procedure STOP_ALL_TIMERS;
  procedure START_ALL_TIMERS;
  procedure SUBTRACT_HOST_TIME(T: duration; NAME: TIMER);
   -- Subtract T (host machine duration) from the reading on the timer
  procedure SUBTRACT_HOST_TIME_FROM_ALL_TIMERS(T: duration);
   -- Subtract T (host machine duration) from the reading on all timers

  pragma INLINE(READ, RESET, START, STOP);
  pragma INLINE(HOST_DURATION, STOP_ALL_TIMERS,
START_ALL_TIMERS);

private
  type timer_list_record is
    record
      value: TIMER;
      next: timer_list;
    end record;
  type timer_list is access timer_list_record;

  empty_timer_list: constant timer_list := null;
  type STATE is (RUNNING, STOPPED);

  type TIMER_RECORD is
    record
      -- All times in a TIMER_RECORD are wrt the caps host machine.
      START_TIME: TIME;  -- Meaningful only if PRESENT_STATE = RUNNING.
      ELAPSED_TIME: DURATION := 0.0;
      PRESENT_STATE: STATE := STOPPED;
    end record;
  type TIMER is access TIMER_RECORD;

end RUN_TIME_TIMER;


package RUN_TIME_RESULTS is

LOGFILE_IS_OPEN : boolean := FALSE;

  procedure
GET_OPERATOR_RUN_TIME_CALCULATION_DATA(Operator_Data:in out
RUN_TIME_RECORD);

  procedure SEND_EXECUTE_RUN_TIME_DATA;
```

```
  procedure Open_Log_File(Log_File : in out File_Type; Log_Mode: in File_Mode;
                          Log_Name: in String; Log_Form: in String);

  procedure Build_Log_File (Log_File: in out File_Type; Log_Mode: in File_Mode;
                          Log_Name: in String; Log_Form: in String);

  procedure Write_Log_File(Log_File: in File_Type; Log_Item: in String;
                                  Log_Data: float);

  procedure Close_Log_File(Log_File: in out File_Type);

end RUN_TIME_RESULTS;
```

F. SOURCE CODE

```
--
*****************************************************************************
-- RUN_TIME_MEASURE.ADS
--
-- This module is called by the main program to monitor the run time of
-- the CAPS operators which have been previously scheduler by the static
-- scheduler
--
--
*****************************************************************************

with TEXT_IO;   use TEXT_IO;
with ADA.REAL_TIME; use ADA.REAL_TIME:


-- Run Time Measure procedures declaration

package RUN_TIME_MEASURE is

  -- Record of operator data
  -- All of the record elements are per instance of each operator.
  type RUN_TIME_RECORD is
    record
      Operator_Name       : String(1..6);
      Execution_Start     : float := 0.0;
      Execution_Stop      : float := 0.0;
      Planned_Start       : float := 0.0;
      Planned_Stop        : float := 0.0;
      Actual_Run_Time     : float := 0.0;
      Planned_Run_Time    : float := 0.0;
      Overhead            : float := 0.0;
      Total_Run_Time      : float := 0.0;
      Total_Timing_Error  : float := 0.0;
      Average_Timing_Error : float := 0.0;
      Average_Run_Time    : float := 0.0;
      Minimum_Run_Time    : float := 100.0;
      Maximum_Run_Time    : float := 0.0;
      Execution_Count     : float := 0.0;
      Run_Time_Difference : float := 0.0;
    end record;
  type RUN_TIME_LIST is access RUN_TIME_RECORD;
```

```
  procedure CREATE_NEW_RT_TIMER;

  procedure START_RT_TIMER;

  procedure RESET_RT_TIMER;

  procedure GET_OPERATOR_EXECUTION_START(
                              Operator_Data: in out RUN_TIME_RECORD);

  procedure GET_OPERATOR_EXECUTION_STOP(
                              Operator_Data: in out RUN_TIME_RECORD);

  procedure SUBTRACT_TIMER(sub_time : in Duration);

  procedure SEND_OPERATOR_EXECUTION_DATA(
                              Operator_Data: in out RUN_TIME_RECORD);


end RUN_TIME_MEASURE;


--*****************************************************************
-- RUN_TIME_MEASURE.ADB
--
-- This module is called by the main program to monitor the run time of
-- the CAPS operators which have been previously scheduler by the static
-- scheduler
--
--*****************************************************************

with TEXT_IO;
with RUN_TIME_TIMER; use RUN_TIME_TIMER;
with RUN_TIME_ANALYSIS; use RUN_TIME_ANALYSIS;
with ADA.REAL_TIME; use ADA.REAL_TIME;


package body RUN_TIME_MEASURE is


  package FLOAT_IO is new TEXT_IO.FLOAT_IO(FLOAT);


  --The real-time timer
```

```
 RT_timer : RUN_TIME_TIMER.TIMER;

--*********************************************************************
-- CREATE NEW TIMER
--
-- This procedure is called by the scheduler program to create a new
-- real time timer by calling the RUN_TIME_TIMER package.
--
--*********************************************************************
 procedure CREATE_NEW_RT_TIMER is

 begin

   -- Call RUN_TIME_TIMER module to create new timer

   RT_timer := RUN_TIME_TIMER.NEW_TIMER;

 end CREATE_NEW_RT_TIMER;


--*********************************************************************
-- START_RT_TIMER
--
-- This procedure is called by the scheduler program to start the real
-- time timer operating.
--
--*********************************************************************
 procedure START_RT_TIMER is

 begin


   -- Call RUN_TIME_TIMER module to start the real-time timer

   RUN_TIME_TIMER.START(RT_timer);

 end START_RT_TIMER;

--*********************************************************************
-- RESET_RT_TIMER
--
-- This procedure is called by the scheduler program to perform a reset
-- of the real time timer.
--*********************************************************************
```

```
procedure RESET_RT_TIMER is

begin

  -- Call RUN_TIME_TIMER module for counter reset

  RUN_TIME_TIMER.RESET(RT_timer);

end RESET_RT_TIMER;


--*********************************************************************
-- GET_OPERATOR_EXECUTION_START
--
-- This procedure is called by the scheduler program to retrieve the
-- actual execution starting time of the CAPS operator.
--
--*********************************************************************
 procedure GET_OPERATOR_EXECUTION_START(
                Operator_Data: in out RUN_TIME_RECORD) is

 begin

  -- Initialize start time
  Operator_Data.Execution_Start := 0.0;


  -- Get the real-time timer counter
  Operator_Data.Execution_Start :=
      float(RUN_TIME_TIMER.HOST_DURATION(RT_timer));

 end GET_OPERATOR_EXECUTION_START;


--*********************************************************************
-- GET_OPERATOR_EXECUTION_STOP
--
-- This procedure is called by the scheduler program to retrieve the
-- actual execution stopping time of the CAPS operator.
--
--*********************************************************************
 procedure GET_OPERATOR_EXECUTION_STOP(
                    Operator_Data: in out RUN_TIME_RECORD) is
```

```
 begin

   -- Initialize stop time
   Operator_Data.Execution_Stop  := 0.0;

   -- Get the real-time timer counter
   Operator_Data.Execution_Stop  :=
       float(RUN_TIME_TIMER.HOST_DURATION(RT_timer));

   -- Update the execution count
   Operator_Data.Execution_Count := Operator_Data.Execution_Count + 1.0;

 end GET_OPERATOR_EXECUTION_STOP;


--*********************************************************************
-- SUBTRACT_TIMER
--
-- This procedure is called within the scheduler program to subtract
-- time from the real-time timer. This allows for alignment to actual
-- execution time with the CAPS operator.
--
--*********************************************************************
 procedure SUBTRACT_TIMER(sub_time : in Duration) is

 begin

    RUN_TIME_TIMER.SUBTRACT_HOST_TIME(
          RUN_TIME_TIMER.HOST_DURATION(RT_timer) - sub_time, RT_timer );

 end SUBTRACT_TIMER;

--*********************************************************************
-- SEND_OPERATOR_EXECUTION_DATA
--
-- This procedure is called within the scheduler program to transmit the
-- actual execution start and stop time of the CAPS operator to the
-- RUN_TIME_ANALYSIS module.
--
--*********************************************************************
 procedure SEND_OPERATOR_EXECUTION_DATA(
                          Operator_Data:in out RUN_TIME_RECORD) is

 begin
```

```
   --Send execution data to RUN_TIME_ANALYSIS

   GET_OPERATOR_EXECUTION_DATA(Operator_Data);


  end SEND_OPERATOR_EXECUTION_DATA;


end RUN_TIME_MEASURE;

==============================

--*********************************************************************
-- RUN_TIME_TIMER.ADS
--
-- This module is called by the RUN_TIME_MEASURE module to create and
-- manipulate real-time timers.  This package is patterned after
-- the PSDL_TIMERS package to allow ease of future transition from
-- ADA.CALENDER to ADA.REAL_TIME.
--
--*********************************************************************
with TEXT_IO;   use TEXT_IO;
with Ada.Real_Time; use Ada.Real_Time;

package RUN_TIME_TIMER is

subtype MILLISEC is NATURAL;
type TIMER is private;
type timer_list is private;

  empty_timer_list: constant timer_list;
  function first(tl: timer_list) return TIMER; -- raises no_subcomponents
  function rest(tl: timer_list) return timer_list; -- raises no_subcomponents
  procedure add(t: TIMER; tl: in out timer_list);

  no_subcomponents: exception;

  function READ(NAME: in TIMER) return MILLISEC;
    -- Timer reading wrt the target machine.
  procedure RESET(NAME: TIMER);
  procedure START(NAME: TIMER);
  procedure STOP(NAME: TIMER);

  -- operations used by the CAPS tools.
```

```
  function NEW_TIMER return TIMER; -- creates and initializes a new timer
  function HOST_DURATION(NAME: TIMER) return duration;
    -- total time accumulated in the Timer on the host machine.

  function TARGET_TO_HOST(d: DURATION) return DURATION;
    -- Converts durations on the target machine to the
    -- corresponding durations on the CAPS host machine.
  procedure STOP_ALL_TIMERS;
  procedure START_ALL_TIMERS;
  procedure SUBTRACT_HOST_TIME(T: duration; NAME: TIMER);
   -- Subtract T (host machine duration) from the reading on the timer
  procedure SUBTRACT_HOST_TIME_FROM_ALL_TIMERS(T: duration);
   -- Subtract T (host machine duration) from the reading on all timers

  pragma INLINE(READ, RESET, START, STOP);
  pragma INLINE(HOST_DURATION, STOP_ALL_TIMERS,
START_ALL_TIMERS);

private
  type timer_list_record is
    record
      value: TIMER;
      next: timer_list;
    end record;
  type timer_list is access timer_list_record;
  empty_timer_list: constant timer_list := null;
  type STATE is (RUNNING, STOPPED);
  type TIMER_RECORD is
    record
      -- All times in a TIMER_RECORD are wrt the caps host machine.
      START_TIME: TIME;  -- Meaningful only if PRESENT_STATE = RUNNING.
      ELAPSED_TIME: DURATION := 0.0;
      PRESENT_STATE: STATE := STOPPED;
    end record;
  type TIMER is access TIMER_RECORD;

end RUN_TIME_TIMER;



--*****************************************************************
-- RUN_TIME_TIMER.ADB
--
-- This module is called by the RUN_TIME_MEASURE module to create and
-- manipulate real-time timers.  This package is patterned after
```

```
-- the PSDL_TIMERS package to allow ease of future transition from
-- ADA.CALENDER to ADA.REAL_TIME.
--
--*********************************************************************

with Ada.Real_Time; use Ada.Real_Time;
with CAPS_HARDWARE_MODEL; use CAPS_HARDWARE_MODEL;

package body RUN_TIME_TIMER is

-- Real time timer list functions

--*********************************************************************
-- FIRST
--
-- This function is called to find first timer in timer list
--
--*********************************************************************
 function first(tl: timer_list) return TIMER is

 begin

   if tl = empty_timer_list then
     raise no_subcomponents;

   else
     return tl.value;

   end if;

 end first;

--*********************************************************************
-- REST
--
-- This function finds the next timer in the timer list
--
--*********************************************************************
 function rest(tl: timer_list) return timer_list is

 begin

   if tl = empty_timer_list then
     raise no_subcomponents;
```

```
    else
      return tl.next;

    end if;

  end rest;

--*********************************************************************
-- ADD
--
-- This procedure places a timer into the timer list
--
--*********************************************************************
  procedure add(t: TIMER; tl: in out timer_list) is

  begin

    tl := new timer_list_record'(value => t, next => tl);

  end add;


-- Real time timer manipulation functions

  -- A list containing all the timers in the prototype and schedules.
  timers: timer_list := empty_timer_list;

--*********************************************************************
-- CONVERT_TO_TARGET_TIME
--
-- This function is called to provide a targeted host time calculation.
-- The calculation converts elapsed time to milliseconds.
--
--*********************************************************************
function CONVERT_TO_TARGET_TIME(d: DURATION) return MILLISEC is

    CONVERSION_FACTOR: constant FLOAT:= 1000.0;

  begin

    return MILLISEC(FLOAT(d) * CONVERSION_FACTOR / CPU_SPEED_RATIO);

  end CONVERT_TO_TARGET_TIME;
```

```
--*********************************************************************
-- TARGET_TO_HOST
--
-- This function is called to adapt the time duration to a targeted
-- host CPU speed. The function converts durations on the target
-- machine to the corresponding durations on the CAPS host machine.
--
--*********************************************************************
function TARGET_TO_HOST(d: DURATION) return DURATION is

 begin

   return  DURATION(FLOAT(d) * CPU_SPEED_RATIO);

 end TARGET_TO_HOST;


--*********************************************************************
-- READ
--
-- This function is called to read the clock counter.
--
--*********************************************************************
 function READ(NAME: in TIMER) return MILLISEC is

    --JD TO USE SPLIT procedure in a-reatim.adb
    split_time  : time_span;
    split_time1 : time_span;
    split_time2 : time_span;
    timer_time  : TIME;
    timer_time1 : TIME;
    output_time : duration := 0.0;
    seconds     : Seconds_Count;

 begin

   case NAME.PRESENT_STATE is
    when RUNNING =>
       --CHG Elapsed_time from Duration to Time_Span Type
       split_time := to_time_span(NAME.ELAPSED_TIME);

       --GET the first part of convert-to-target-time call
       timer_time := split_time + CLOCK;
       ADA.REAL_TIME.split(NAME.START_TIME,seconds,split_time1);
```

```
        --GET the second part of convert-to-target-time call
        timer_time1 := timer_time - split_time1;

        --Send the result of the read
        ADA.REAL_TIME.split(timer_time1,seconds,split_time2);
        output_time := to_duration(split_time2);

        return CONVERT_TO_TARGET_TIME(output_time);

     when STOPPED => return
CONVERT_TO_TARGET_TIME(NAME.ELAPSED_TIME);
   end case;
 end READ;


--*********************************************************************
-- RESET
--
-- This procedure is called to reset the clock counter to zero.
--
--*********************************************************************
 procedure RESET(NAME: TIMER) is
 begin

   NAME.ELAPSED_TIME:= 0.0;

   case NAME.PRESENT_STATE is

     when RUNNING => NAME.START_TIME := CLOCK;

     when STOPPED => null;

   end case;

 end RESET;


--*********************************************************************
-- START
--
-- This function is called to run the timer counter.
--
--*********************************************************************
 procedure START(NAME: TIMER) is
```

```
 begin

  case NAME.PRESENT_STATE is

    when RUNNING => null;

    when STOPPED =>

      NAME.PRESENT_STATE:= RUNNING;

      NAME.START_TIME:= CLOCK;

  end case;

 end START;



--*********************************************************************
-- STOP
--
-- This function is called to halt the clock counter
--
--*********************************************************************
 procedure STOP(NAME: TIMER) is

    --JD TO USE SPLIT procedure in a-reatim.adb
    split_time  : time_span;
    split_time1 : time_span;
    split_time2 : time_span;
    timer_time  : TIME;
    timer_time1 : TIME;
    output_time : duration;
    seconds     : Seconds_Count;

 begin

   case NAME.PRESENT_STATE is
     when RUNNING =>

       NAME.PRESENT_STATE:= STOPPED;

         --CHG Elapsed_time from Duration to Time_Span Type
         split_time := to_time_span(NAME.ELAPSED_TIME);
```

```
        --GET the first part of elapsed time calculation
        timer_time := split_time + CLOCK;
        ADA.REAL_TIME.split(NAME.START_TIME,seconds,split_time1);

        --GET the second part of elapsed time calculation
        timer_time1 := timer_time - split_time1;
        ADA.REAL_TIME.split(timer_time1,seconds,split_time2);
        output_time := to_duration(split_time2);

        --Assign the result of the counter
        NAME.ELAPSED_TIME := output_time;

      when STOPPED => null;

    end case;

  end STOP;

--*********************************************************************
-- NEW_TIMER
--
-- This function is called to
-- create and initializes a new timer
--
--*********************************************************************
  function NEW_TIMER return TIMER is

    result: timer;

  begin

    result := new TIMER_RECORD;

    add(result, timers);

    return result;

  end NEW_TIMER;


--*********************************************************************
-- HOST_DURATION
--
-- This function when called provides the
-- total time accumulated in the Timer on the host machine,
```

```
-- used instead of the real-time clock in the static schedule
--
--*****************************************************************
 function HOST_DURATION(NAME: TIMER) return duration is

   --SPLIT procedure in a-reatim.adb
   split_time  : time_span;
   split_time1 : time_span;
   split_time2 : time_span:
   timer_time  : TIME;
   timer_time1 : TIME:
   output_time : duration;
   seconds     : Seconds_Count;
   seconds1    : Seconds_Count;

 begin


   case NAME.PRESENT_STATE is


     when RUNNING =>

       --CHG Elapsed_time from Duration to Time_Span Type
       split_time := to_time_span(NAME.ELAPSED_TIME);

       --GET the first part of elapsed time calculation
       timer_time := split_time + CLOCK;
       ADA.REAL_TIME.split(NAME.START_TIME,seconds,split_time1);

       --GET the second part of elapsed time calculation
       timer_time1 := timer_time - split_time1;
       ADA.REAL_TIME.split(timer_time1,seconds1,split_time2);
       output_time := to_duration(split_time2);

       --Send the result of the elapsed time
       return (output_time);

     when STOPPED => return (NAME.ELAPSED_TIME);

   end case;

 end HOST_DURATION;
```

```
--*********************************************************************
-- STOP_ALL_TIMERS
--
-- This procedure when called provides the a stop point for all timers in
-- the timer list.
--
--*********************************************************************
 procedure STOP_ALL_TIMERS is

   tl: timer_list := timers;

 begin

   while tl /= empty_timer_list loop

     STOP(first(tl));

     tl := rest(tl);

   end loop;

 end STOP_ALL_TIMERS:


--*********************************************************************
-- START_ALL_TIMERS
--
-- This procedure when called provides the a start point for all timers in
-- the timer list.
--
--*********************************************************************
 procedure START_ALL_TIMERS is

   tl: timer_list := timers;

 begin

   while tl /= empty_timer_list loop

     START(first(tl));

     tl := rest(tl);
   end loop;
 end START_ALL_TIMERS;
```

```
--*****************************************************************
-- SUBTRACT_HOST_TIME
--
-- This procedure when called provides subratction of time from timer
-- Subtract T (host machine duration) from the reading on timer NAME
--
--*****************************************************************
 procedure SUBTRACT_HOST_TIME(T: duration; NAME: TIMER) is

 begin

  NAME.ELAPSED_TIME := NAME.ELAPSED_TIME - T:

 end SUBTRACT_HOST_TIME;



--*****************************************************************
-- SUBTRACT_HOST_TIME_FROM_ALL_TIMERS
--
-- This procedure when called provides subratction of time from timers
-- in timer list. Subtract T (host machine duration) from the reading
-- on timer NAME
--
--*****************************************************************
 procedure SUBTRACT_HOST_TIME_FROM_ALL_TIMERS(T: duration) is

  tl: timer_list := timers;

 begin

  while tl /= empty_timer_list loop

   SUBTRACT_HOST_TIME(T, first(tl));

   tl := rest(tl);

  end loop;

 end SUBTRACT_HOST_TIME_FROM_ALL_TIMERS;

end RUN_TIME_TIMER;

==============================
```

```
--*****************************************************************
-- RUN_TIME_ANALYSIS.ADS
--
-- This module is called to perform an analysis upon the CAPS operator
-- execution run time.
--
--*****************************************************************
with TEXT_IO;  use TEXT_IO;
with RUN_TIME_MEASURE;  use RUN_TIME_MEASURE;

package RUN_TIME_ANALYSIS is

-- Run Time Analysis procedures declaration

 procedure GET_OPERATOR_EXECUTION_DATA(
                              Operator_Data : in out RUN_TIME_RECORD);

 procedure GET_OPERATOR_TIMING_DATA(
                              Operator_Data : in out RUN_TIME_RECORD);

 procedure ANALYZE_RESULTS_DATA(
                              Operator_Data : in out RUN_TIME_RECORD);

 procedure SEND_OPERATOR_RUN_TIME_CALCULATION_DATA(
                              Operator_Data:in out RUN_TIME_RECORD);


end RUN_TIME_ANALYSIS;


--*****************************************************************
-- RUN_TIME_ANALYSIS.ADB
--
-- This module is called to perform an analysis upon the CAPS operator
-- execution run time.
--
--*****************************************************************

with TEXT_IO;
with RUN_TIME_MEASURE; use RUN_TIME_MEASURE;
with RUN_TIME_RESULTS; use RUN_TIME_RESULTS;
```

```
package body RUN_TIME_ANALYSIS is

  package FLOAT_IO is new TEXT_IO.FLOAT_IO(FLOAT);



--*********************************************************************
-- GET_OPERATOR_EXECUTION_DATA
--
-- This procedure is called by the scheduler program to get actual run
-- time data from RUN_TIME_MEASURE module. This run time data includes
-- actual execution starting and stopping time of the CAPS operator.
--
--*********************************************************************
 procedure GET_OPERATOR_EXECUTION_DATA(
                                Operator_Data: in out RUN_TIME_RECORD) is

  begin

    -- Initialize data variable
    Operator_Data.Actual_Run_Time := 0.0;

   if (Operator_Data.Execution_Stop > Operator_Data.Execution_Start) then

   -- Calculate Actual Run Time
     Operator_Data.Actual_Run_Time :=
                    Operator_Data.Execution_Stop - Operator_Data.Execution_Start;
   else

   -- Raise Exception
null;
--     PUT_LINE("RUN_TIME_ANALYSIS: GET_OPERATOR_EXECUTION_DATA:
STOP < START");

   end if;

   -- Send results to be analyzed
   ANALYZE_RESULTS_DATA(Operator_Data);


  end GET_OPERATOR_EXECUTION_DATA;
```

```
--
*************************************************************************
-- GET_OPERATOR_TIMING_DATA
--
-- This procedure is called by the scheduler program to get the planned
-- run time data from CAPS SCHEDULER module. This scheduled run time data
-- includes predetermined execution starting and stopping time of the
-- CAPS operator.
--
--
*************************************************************************
 procedure GET_OPERATOR_TIMING_DATA(Operator_Data : in out
RUN_TIME_RECORD) is


 begin

  -- Initialize data variable
  Operator_Data.Planned_Run_Time := 0.0;

 if (Operator_Data.Planned_Stop > Operator_Data.Planned_Start) then

   -- Calculate Planned Run Time
   Operator_Data.Planned_Run_Time :=
          (Operator_Data.Planned_Stop - Operator_Data.Planned_Start);
 else
   PUT_LINE("RUN_TIME_ANALYSIS:
                    GET_OPERATOR_TIMING_DATA: STOP < START");

 end if;


 end GET_OPERATOR_TIMING_DATA;


--*************************************************************************
-- ANALYZE_RESULTS_DATA
--
-- This procedure is called within the RUN_TIME_ANALYSIS module to
-- perform an analysis upon the recently retrived CAPS operator
-- run time data. This run time data consists of the planned execution
-- starting and stopping, as well as actual execution start and stop
-- times of this CAPS operator.
```

```
--
--*****************************************************************
procedure ANALYZE_RESULTS_DATA(
                                    Operator_Data : in out RUN_TIME_RECORD) is


 begin

 -- Get total run time
 Operator_Data.Total_Run_Time :=
     Operator_Data.Actual_Run_Time + Operator_Data.Total_Run_Time;

 -- Determine average run time
 Operator_Data.Average_Run_Time :=
     (Operator_Data.Total_Run_Time / Operator_Data.Execution_Count):

 -- Check for new minimum run time
 if (Operator_Data.Actual_Run_Time < Operator_Data.Minimum_Run_Time) then

   Operator_Data.Minimum_Run_Time := Operator_Data.Actual_Run_Time;

 end if;

 -- Check for new maximum run time
 if (Operator_Data.Actual_Run_Time > Operator_Data.Maximum_Run_Time) then

   Operator_Data.Maximum_Run_Time := Operator_Data.Actual_Run_Time;

 end if;

 -- Find planned/actual run time difference
 if (Operator_Data.Actual_Run_Time > Operator_Data.Planned_Run_Time) then

   Operator_Data.Run_Time_Difference :=
     Operator_Data.Actual_Run_Time - Operator_Data.Planned_Run_Time:

 else

   Operator_Data.Run_Time_Difference :=
     Operator_Data.Planned_Run_Time - Operator_Data.Actual_Run_Time;

 end if;


 -- Ship execution data
```

```
 SEND_OPERATOR_RUN_TIME_CALCULATION_DATA(Operator_Data);

 end ANALYZE_RESULTS_DATA;


--*********************************************************************
-- SEND_OPERATOR_RUN_TIME_CALCULATION_DATA
--
-- This procedure is called by the scheduler program to
-- transmit run time calculations to RUN_TIME_RESULTS module
-- This scheduled run time data includes predetermined execution starting
-- and stopping time of the CAPS operator as well as actual execution
-- start and stop times of this operator.
--
--*********************************************************************
 procedure SEND_OPERATOR_RUN_TIME_CALCULATION_DATA(
                                   Operator_Data:in out RUN_TIME_RECORD) is

 begin

 -- Send the execution data to RUN_TIME_RESULTS module

 GET_OPERATOR_RUN_TIME_CALCULATION_DATA(Operator_Data);

 end SEND_OPERATOR_RUN_TIME_CALCULATION_DATA;


end RUN_TIME_ANALYSIS;

==============================

--*********************************************************************
-- RUN_TIME_RESULTS.ADS
--
-- This module is called to provide the results of the run time
-- measurement and analysis of the CAPS operators which have
-- been previously performed within the CAPS environment.
-- The resultant run time statistics are stored within an execution
-- logfile ("LOGFILE").
--
--*********************************************************************

with TEXT_IO;   use TEXT_IO;
with RUN_TIME_ANALYSIS; use RUN_TIME_ANALYSIS;
```

```
with RUN_TIME_MEASURE: use RUN_TIME_MEASURE;

package RUN_TIME_RESULTS is

--LOGFILE_IS_OPEN : boolean := FALSE;

 procedure GET_OPERATOR_RUN_TIME_CALCULATION_DATA(Operator_Data:
                                                  in out RUN_TIME_RECORD);

 procedure SEND_EXECUTE_RUN_TIME_DATA;

 procedure Open_Log_File(Log_File : in out File_Type; Log_Mode: in File_Mode;
                Log_Name: in String; Log_Form: in String);

 procedure Build_Log_File (Log_File: in out File_Type; Log_Mode: in File_Mode;
                  Log_Name: in String; Log_Form: in String);

 procedure Write_Log_File(Log_File: in File_Type; Log_Item: in String;
                          Log_Data: float);

 procedure Close_Log_File(Log_File: in out File_Type);


end RUN_TIME_RESULTS;


--*********************************************************************
-- RUN_TIME_RESULTS.ADB
--
-- This module is called by the main program to provide the results of
-- the run time measurement and analysis of the CAPS operators which have
-- been previously performed within the CAPS environment.
-- The resultant run time statistics are stored within an execution
-- logfile ("LOGFILE").
--
--*********************************************************************

with TEXT_IO; use TEXT_IO;
with RUN_TIME_ANALYSIS; use RUN_TIME_ANALYSIS;

package body RUN_TIME_RESULTS is


  --FOR LOGFILE FLOAT IO
  package NEW_FLOAT_IO is new FLOAT_IO(FLOAT);
```

```
use NEW_FLOAT_IO:

package fl_io is new float_io(float);
package FLOAT_IO is new TEXT_IO.FLOAT_IO(FLOAT);


Log_File: File_Type:
Log_Mode: File_Mode := Append_File; --Out_File; --Inout_File;
Log_Name: String := "LOGFILE";
Log_Form: String := "LOG_FORM";
Log_Item: float  := 0.1;



--************************************************************************
-- SEND_EXECUTE_RUN_TIME_DATA
--
-- This procedure is called by the RUN_TIME_ANALYSIS module to get the planned
-- run time data from CAPS SCHEDULER module. This scheduled run time data
-- includes predetermined execution starting and stopping time of the
-- CAPS operator.
--
--************************************************************************
procedure SEND_EXECUTE_RUN_TIME_DATA is


begin

PUT_LINE("RUN_TIME_RESULTS: SEND_EXECUTE_RUN_TIME_DATA:");

--PUT_LINE(" ");
--PUT("EVAL TEMP ACTUAL VS PLANNED TIME:");
----fl_io.put((FLOAT(EVAL_TEMP_RT_ACTUAL_STOP)-
FLOAT(EVAL_TEMP_RT_ACTUAL_START))-
(FLOAT(Evaluate_Temp_STOP_TIME2)-(FLOAT(Evaluate_Temp_START_TIME2)));
--PUT_LINE(" ");

--
end SEND_EXECUTE_RUN_TIME_DATA;
```

```
--*********************************************************************
-- GET_OPERATOR_RUN_TIME_CALCULATION_DATA
--
-- This procedure is called by the scheduler program to get the planned
-- and actual run time data from CAPS SCHEDULER module. This scheduled
-- run time data includes predetermined execution starting and stopping
-- times of the CAPS operator as well as the actual start and stop times.
--
--*********************************************************************
 procedure GET_OPERATOR_RUN_TIME_CALCULATION_DATA(
                              Operator_Data:  in out RUN_TIME_RECORD) is

 begin


   Open_Log_File(Log_File,Log_Mode, Log_Name, Log_Form);

SEND_EXECUTE_RUN_TIME_DATA;

 end GET_OPERATOR_RUN_TIME_CALCULATION_DATA;


--*********************************************************************
-- BUILD_LOG_FILE
--
-- This procedure is called to create a file for logging execution time
-- data. This data will include the scheduled predetermined execution
-- starting and stopping times of the CAPS operator as well as the actual
-- start and stop times.
--
--*********************************************************************
 procedure Build_Log_File (Log_File: in out File_Type; Log_Mode: in File_Mode:
                  Log_Name: in String; Log_Form: in String) is

 begin


 Create(Log_File, Log_Mode, Log_Name, Log_Form);

 Put_Line(Log_File," ");
 Put_Line(Log_File,"RUN TIME EXECUTION MONITOR LOG");
 Put_Line(Log_File," ");


end Build_Log_File;
```

```
--*********************************************************************
-- OPEN_LOG_FILE
--
-- This procedure is called to open a previously created file for logging
-- execution time data. This data includes CAPS operator starting and
-- stopping times.
--
--*********************************************************************
  procedure Open_Log_File(Log_File : in out File_Type; Log_Mode: in File_Mode;
                  Log_Name: in String; Log_Form: in String) is

  begin

    Open(Log_File, Log_Mode, Log_Name, Log_Form);

--if not (Is_Open(Log_File)) then
--Open_Log_File(Log_File,Log_Mode, Log_Name,Log_Form);
--end if:

    if Is_Open(Log_File) then
      --PUT_LINE("RT_RESULTS: Open_Log_File");
      null:
    else
      --PUT_LINE("RT_RESULTS: NOT Open_Log_File");
      null:
    end if;


end Open_Log_File;



--*********************************************************************
-- CLOSE_LOG_FILE
--
-- This procedure is called to close a previously opened file for logging
-- execution time data. This data includes CAPS operator starting and
-- stopping times.
--
--*********************************************************************
  procedure Close_Log_File(Log_File: in out File_Type) is

  begin
```

```
   Close(Log_File);

 end Close_Log_File;


--*********************************************************************
-- WRITE_LOG_FILE
--
-- This procedure is called to write to a previously created and opened
-- file for logging execution time data. This data includes CAPS operator
-- starting and stopping times.
--
--*********************************************************************
 procedure Write_Log_File(Log_File: in File_Type:
                          Log_Item: in String; Log_Data: float) is

 begin

   Put_Line(Log_File,Log_Item);

   NEW_FLOAT_IO.PUT(Log_File,float(Log_Data),0);

   Put_Line(Log_File," ");


end Write_Log_File;


end RUN_TIME_RESULTS;

==============================
```

G. PROTOTYPE CODE

```
with TEMP_CONTROLLER_DRIVERS;  use TEMP_CONTROLLER_DRIVERS:
with PRIORITY_DEFINITIONS;  use PRIORITY_DEFINITIONS;
with PSDL_TIMERS;  use PSDL_TIMERS:
with TEXT_IO;  use TEXT_IO;

with RUN_TIME_MEASURE; use RUN_TIME_MEASURE;
with RUN_TIME_ANALYSIS; use RUN_TIME_ANALYSIS;
--with RUN_TIME_RESULTS; use RUN_TIME_RESULTS;
with RUN_TIME_TIMER; use RUN_TIME_TIMER;

with Ada.Real_Time; use Ada.Real_Time;
with System.Time_Operations;
--Used for Clock

package body TEMP_CONTROLLER_STATIC_SCHEDULERS is

package fl_io is new float_io(float);

package FLOAT_IO is new TEXT_IO.FLOAT_IO(FLOAT);
package INT_IO is new TEXT_IO.INTEGER_IO(INTEGER);

 task type STATIC_SCHEDULE_TYPE is
  pragma priority (STATIC_SCHEDULE_PRIORITY);
  entry START;
 end STATIC_SCHEDULE_TYPE;
 for STATIC_SCHEDULE_TYPE'STORAGE_SIZE use 200_000;
 STATIC_SCHEDULE : STATIC_SCHEDULE_TYPE:

 task body STATIC_SCHEDULE_TYPE is
  PERIOD : duration;
  Sensor_START_TIME1 : duration;
  Sensor_STOP_TIME1 : duration;
  Evaluate_Temp_START_TIME2 : duration;
  Evaluate_Temp_STOP_TIME2 : duration;
  schedule_timer : PSDL_TIMERS.TIMER := PSDL_TIMERS.NEW_TIMER;

 -- Establish array for multiple operator data records
 Max : constant integer := 100;
 type RUN_TIME_ARRAY is array (1 .. Max) of RUN_TIME_RECORD;

 --Establish the operator data records
```

```
   Sensor_Operator_Data: RUN_TIME_ARRAY;
   Evaluate_Temp_Operator_Data: RUN_TIME_ARRAY;

   begin

    accept START;

      PERIOD := PSDL_TIMERS.TARGET_TO_HOST(
                                        duration( 1.00000000000000E-01));

Sensor_START_TIME1:=PSDL_TIMERS.TARGET_TO_HOST(
                                        duration( 0.00000000000000E+00));
--1st Sensor_STOP_TIME1:=PSDL_TIMERS.TARGET_TO_HOST(
                                        duration(1.75000000000000E-01));
--2nd Sensor_STOP_TIME1:=PSDL_TIMERS.TARGET_TO_HOST(
                                        duration(4.50000000000000E-05));
--3rd
Sensor_STOP_TIME1:=PSDL_TIMERS.TARGET_TO_HOST(
                                        duration(1.26000000000000E-04));
--1st Evaluate_Temp_START_TIME2 :=
PSDL_TIMERS.TARGET_TO_HOST(duration( 1.75000000000000E-01));
--2nd Evaluate_Temp_START_TIME2 :=
PSDL_TIMERS.TARGET_TO_HOST(duration( 4.50000000000000E-05));
--3rd
Evaluate_Temp_START_TIME2 :=
PSDL_TIMERS.TARGET_TO_HOST(duration(1.26000000000000E-04));
--1st Evaluate_Temp_STOP_TIME2 := PSDL_TIMERS.TARGET_TO_HOST(
                                        duration( 3.75000000000000E-01));
--2nd Evaluate_Temp_STOP_TIME2 :=
PSDL_TIMERS.TARGET_TO_HOST(duration(1.03700000000000E-03));
--3rd
Evaluate_Temp_STOP_TIME2 :=
PSDL_TIMERS.TARGET_TO_HOST(duration(1.76900000000000E-03));

    --Setup SENSOR operator
    Sensor_Operator_Data(1).Planned_Start := float(Sensor_START_TIME1);
    Sensor_Operator_Data(1).Planned_Stop  := float(Sensor_STOP_TIME1);
    Sensor_Operator_Data(1).Operator_Name := "SENSOR";
    GET_OPERATOR_TIMING_DATA(Sensor_Operator_Data(1));

    --Setup EVALUATE_TEMP operator
    Evaluate_Temp_Operator_Data(1).Planned_Start  :=
                        float(Evaluate_Temp_START_TIME2);
    Evaluate_Temp_Operator_Data(1).Planned_Stop   :=
                        float(Evaluate_Temp_STOP_TIME2);
```

```
    Evaluate_Temp_Operator_Data(1).Operator_Name  := "EVAL  ";
    GET_OPERATOR_TIMING_DATA(Evaluate_Temp_Operator_Data(1));

    --Create the real-time timer
    CREATE_NEW_RT_TIMER;

    --Initialize the real-time timer
    RUN_TIME_MEASURE.START_RT_TIMER;

    START(schedule_timer);

    loop

    if (Sensor_START_TIME1 > PSDL_TIMERS.HOST_DURATION(schedule_timer))
then
      delay(Sensor_START_TIME1 -
PSDL_TIMERS.HOST_DURATION(schedule_timer));
    end if;


      --Stop/start scheduler timer and get actual execution times
      PSDL_TIMERS.STOP_ALL_TIMERS;
      GET_OPERATOR_EXECUTION_START(Sensor_Operator_Data(1));
      PSDL_TIMERS.START_ALL_TIMERS;

      Sensor_DRIVER;

      --Stop the scheduler timer and get actual execution times
      PSDL_TIMERS.STOP_ALL_TIMERS;
      GET_OPERATOR_EXECUTION_STOP(Sensor_Operator_Data(1));
      PSDL_TIMERS.START_ALL_TIMERS;

      if PSDL_TIMERS.HOST_DURATION(schedule_timer) > Sensor_STOP_TIME1
 then
        PUT_LINE("timing error from operator Sensor");

        --Stop/start scheduler timer and get actual execution times
        PSDL_TIMERS.STOP_ALL_TIMERS;

          --Increment timing error totals
        Sensor_Operator_Data(1).Total_Timing_Error :=
                Sensor_Operator_Data(1).Total_Timing_Error + 1.0;
        PSDL_TIMERS.START_ALL_TIMERS;

        --Subtract any slack time from real-time monitor timer
```

```
        RUN_TIME_MEASURE.SUBTRACT_TIMER(
            PSDL_TIMERS.HOST_DURATION(schedule_timer) -
Sensor_STOP_TIME1);


        --Subtract any slack time from scheduler timer
        PSDL_TIMERS.SUBTRACT_HOST_TIME_FROM_ALL_TIMERS(
    PSDL_TIMERS.HOST_DURATION(schedule_timer) - Sensor_STOP_TIME1);

      end if;

if(Evaluate_Temp_START_TIME2 >
PSDL_TIMERS.HOST_DURATION(schedule_timer)) then
  delay(Evaluate_Temp_START_TIME2 -
PSDL_TIMERS.HOST_DURATION(schedule_timer));
end if;

      --Stop/start scheduler timer and get actual execution times
      PSDL_TIMERS.STOP_ALL_TIMERS;
      GET_OPERATOR_EXECUTION_START(Evaluate_Temp_Operator_Data(1));
      PSDL_TIMERS.START_ALL_TIMERS;

      Evaluate_Temp_DRIVER;

      --Stop the scheduler timer and get actual execution times
      PSDL_TIMERS.STOP_ALL_TIMERS;
      GET_OPERATOR_EXECUTION_STOP(Evaluate_Temp_Operator_Data(1));
      PSDL_TIMERS.START_ALL_TIMERS;


   if PSDL_TIMERS.HOST_DURATION(schedule_timer) >
                        Evaluate_Temp_STOP_TIME2 then

        PUT_LINE("timing error from operator Evaluate_Temp");

        --Stop/start seheduler timer and get actual execution times
        PSDL_TIMERS.STOP_ALL_TIMERS;

          --Increment timing error totals
        Evaluate_Temp_Operator_Data(1).Total_Timing_Error :=
                  Evaluate_Temp_Operator_Data(1).Total_Timing_Error + 1.0;

        PSDL_TIMERS.START_ALL_TIMERS;

        --Subtract any slack time from real-time monitor timer
```

```
      RUN_TIME_MEASURE.SUBTRACT_TIMER(
        PSDL_TIMERS.HOST_DURATION(schedule_timer) -
              Evaluate_Temp_STOP_TIME2);

      --Subtract any slack time from scheduler timer
      PSDL_TIMERS.SUBTRACT_HOST_TIME_FROM_ALL_TIMERS(
                  PSDL_TIMERS.HOST_DURATION(schedule_timer)
                  - Evaluate_Temp_STOP_TIME2);

    end if;

    delay(PERIOD - PSDL_TIMERS.HOST_DURATION(schedule_timer));

    --Most intensive overhead stuff here to minimize
      --scheduling interference


    --Stop/start seheduler timer and get actual execution times
    PSDL_TIMERS.STOP_ALL_TIMERS;

    SEND_OPERATOR_EXECUTION_DATA(Sensor_Operator_Data(1));

    SEND_OPERATOR_EXECUTION_DATA(Evaluate_Temp_Operator_Data(1));

    RUN_TIME_MEASURE.RESET_RT_TIMER;

    --Stop/start seheduler timer and get actual execution times
    PSDL_TIMERS.START_ALL_TIMERS;

    RESET(schedule_timer);


   end loop;

  end STATIC_SCHEDULE_TYPE;

  procedure START_STATIC_SCHEDULE is
  begin
    STATIC_SCHEDULE.START;
  end START_STATIC_SCHEDULE;

end TEMP_CONTROLLER_STATIC_SCHEDULERS;
```

H. ANALYSIS LOGFILE

OPERATOR RUN TIME

ANALYSIS LOGFILE

=== 1ST RUN ===

SENSOR

0.00000E+00

EXECUTION COUNT:

1.00000E+00

CURRENT RUN TIME:

3.20000E-04

MAXIMUM RUN TIME:

3.20000E-04

MINIMUM RUN TIME:

3.20000E-04

AVERAGE RUN TIME:

3.20000E-04

PLANNED RUN TIME:

1.75000E-01

TOTAL TIMING ERRORS:

0.00000E+00

EVAL

0.00000E+00

EXECUTION COUNT:

1.00000E+00

CURRENT RUN TIME:

2.07800E-03

MAXIMUM RUN TIME:

2.07800E-03

MINIMUM RUN TIME:

2.07800E-03

AVERAGE RUN TIME:

2.07800E-03

PLANNED RUN TIME:

2.00000E-01

TOTAL TIMING ERRORS:

0.00000E+00

SENSOR

0.00000E+00

EXECUTION COUNT:

2.00000E+00

CURRENT RUN TIME:

8.80000E-05

MAXIMUM RUN TIME:

3.20000E-04

MINIMUM RUN TIME:

8.80000E-05

AVERAGE RUN TIME:

2.04000E-04

PLANNED RUN TIME:

1.75000E-01

TOTAL TIMING ERRORS:

0.00000E+00

EVAL

0.00000E+00

EXECUTION COUNT:

2.00000E+00

CURRENT RUN TIME:

1.19901E-03

MAXIMUM RUN TIME:

2.07800E-03

MINIMUM RUN TIME:

1.19901E-03

AVERAGE RUN TIME:

1.63850E-03

PLANNED RUN TIME:

2.00000E-01

TOTAL TIMING ERRORS:

0.00000E+00

SENSOR

0.00000E+00

EXECUTION COUNT:

3.00000E+00

CURRENT RUN TIME:

9.40000E-05

MAXIMUM RUN TIME:

3.20000E-04

MINIMUM RUN TIME:

8.80000E-05

AVERAGE RUN TIME:

1.67333E-04

PLANNED RUN TIME:

1.75000E-01

TOTAL TIMING ERRORS:

0.00000E+00

EVAL

0.00000E+00

EXECUTION COUNT:

3.00000E+00

CURRENT RUN TIME:

1.29700E-03

MAXIMUM RUN TIME:

2.07800E-03

MINIMUM RUN TIME:

1.19901E-03

AVERAGE RUN TIME:

1.52467E-03

PLANNED RUN TIME:

2.00000E-01

TOTAL TIMING ERRORS:

0.00000E+00

SENSOR

0.00000E+00

EXECUTION COUNT:

4.00000E+00

CURRENT RUN TIME:

9.80000E-05

MAXIMUM RUN TIME:

3.20000E-04

MINIMUM RUN TIME:

8.80000E-05

AVERAGE RUN TIME:

1.50000E-04

PLANNED RUN TIME:

1.75000E-01

TOTAL TIMING ERRORS:

0.00000E+00

EVAL

0.00000E+00

EXECUTION COUNT:

4.00000E+00

CURRENT RUN TIME:

1.29299E-03

MAXIMUM RUN TIME:

2.07800E-03

MINIMUM RUN TIME:

1.19901E-03

AVERAGE RUN TIME:

1.46675E-03

PLANNED RUN TIME:

2.00000E-01

TOTAL TIMING ERRORS:

0.00000E+00

SENSOR

0.00000E+00

EXECUTION COUNT:

5.00000E+00

CURRENT RUN TIME:

9.00000E-05

MAXIMUM RUN TIME:

3.20000E-04

MINIMUM RUN TIME:

8.80000E-05

AVERAGE RUN TIME:

1.38000E-04

PLANNED RUN TIME:

1.75000E-01

TOTAL TIMING ERRORS:

0.00000E+00

EVAL

0.00000E+00

EXECUTION COUNT:

5.00000E+00

CURRENT RUN TIME:

1.20100E-03

MAXIMUM RUN TIME:

2.07800E-03

MINIMUM RUN TIME:

1.19901E-03

AVERAGE RUN TIME:

1.41360E-03

PLANNED RUN TIME:

2.00000E-01

TOTAL TIMING ERRORS:

0.00000E+00

SENSOR

0.00000E+00

EXECUTION COUNT:

6.00000E+00

CURRENT RUN TIME:

9.20000E-05

MAXIMUM RUN TIME:

3.20000E-04

MINIMUM RUN TIME:

8.80000E-05

AVERAGE RUN TIME:

1.30333E-04

PLANNED RUN TIME:

1.75000E-01

TOTAL TIMING ERRORS:

0.00000E+00

EVAL

0.00000E+00

EXECUTION COUNT:

6.00000E+00

CURRENT RUN TIME:

1.41999E-03

MAXIMUM RUN TIME:

2.07800E-03

MINIMUM RUN TIME:

1.19901E-03

AVERAGE RUN TIME:

1.41466E-03

PLANNED RUN TIME:

2.00000E-01

TOTAL TIMING ERRORS:

0.00000E+00

SENSOR

0.00000E+00

EXECUTION COUNT:

7.00000E+00

CURRENT RUN TIME:

8.90000E-05

MAXIMUM RUN TIME:

3.20000E-04

MINIMUM RUN TIME:

8.80000E-05

AVERAGE RUN TIME:

1.24429E-04

PLANNED RUN TIME:

1.75000E-01

TOTAL TIMING ERRORS:

0.00000E+00

EVAL

0.00000E+00

EXECUTION COUNT:

7.00000E+00

CURRENT RUN TIME:

1.31400E-03

MAXIMUM RUN TIME:

2.07800E-03

MINIMUM RUN TIME:

1.19901E-03

AVERAGE RUN TIME:

1.40028E-03

PLANNED RUN TIME:

2.00000E-01

TOTAL TIMING ERRORS:

0.00000E+00

SENSOR

0.00000E+00

EXECUTION COUNT:

8.00000E+00

CURRENT RUN TIME:

9.30000E-05

MAXIMUM RUN TIME:

3.20000E-04

MINIMUM RUN TIME:

8.80000E-05

AVERAGE RUN TIME:

1.20500E-04

PLANNED RUN TIME:

1.75000E-01

TOTAL TIMING ERRORS:

0.00000E+00

EVAL

0.00000E+00

EXECUTION COUNT:

8.00000E+00

CURRENT RUN TIME:

1.20300E-03

MAXIMUM RUN TIME:

2.07800E-03

MINIMUM RUN TIME:

1.19901E-03

AVERAGE RUN TIME:

1.37562E-03

PLANNED RUN TIME:

2.00000E-01

TOTAL TIMING ERRORS:

0.00000E+00

=== 2ND RUN ===

SENSOR

0.00000E+00

EXECUTION COUNT:

1.00000E+00

CURRENT RUN TIME:

2.60000E-04

MAXIMUM RUN TIME:

2.60000E-04

MINIMUM RUN TIME:

2.60000E-04

AVERAGE RUN TIME:

2.60000E-04

PLANNED RUN TIME:

4.50000E-05

TOTAL TIMING ERRORS:

1.00000E+00

EVAL

0.00000E+00

EXECUTION COUNT:

1.00000E+00

CURRENT RUN TIME:

1.56500E-03

MAXIMUM RUN TIME:

1.56500E-03

MINIMUM RUN TIME:

1.56500E-03

AVERAGE RUN TIME:

1.56500E-03

PLANNED RUN TIME:

9.92000E-04

TOTAL TIMING ERRORS:

1.00000E+00

SENSOR

0.00000E+00

EXECUTION COUNT:

2.00000E+00

CURRENT RUN TIME:

9.50000E-05

MAXIMUM RUN TIME:

2.60000E-04

MINIMUM RUN TIME:

9.50000E-05

AVERAGE RUN TIME:

1.77500E-04

PLANNED RUN TIME:

4.50000E-05

TOTAL TIMING ERRORS:

2.00000E+00

EVAL

0.00000E+00

EXECUTION COUNT:

2.00000E+00

CURRENT RUN TIME:

1.16400E-03

MAXIMUM RUN TIME:

1.56500E-03

MINIMUM RUN TIME:

1.16400E-03

AVERAGE RUN TIME:

1.36450E-03

PLANNED RUN TIME:

9.92000E-04

TOTAL TIMING ERRORS:

2.00000E+00

SENSOR

0.00000E+00

EXECUTION COUNT:

3.00000E+00

CURRENT RUN TIME:

9.30000E-05

MAXIMUM RUN TIME:

2.60000E-04

MINIMUM RUN TIME:

9.30000E-05

AVERAGE RUN TIME:

1.49333E-04

PLANNED RUN TIME:

4.50000E-05

TOTAL TIMING ERRORS:

3.00000E+00

EVAL

0.00000E+00

EXECUTION COUNT:

3.00000E+00

CURRENT RUN TIME:

1.15000E-03

MAXIMUM RUN TIME:

1.56500E-03

MINIMUM RUN TIME:

1.15000E-03

AVERAGE RUN TIME:

1.29300E-03

PLANNED RUN TIME:

9.92000E-04

TOTAL TIMING ERRORS:

3.00000E+00

SENSOR

0.00000E+00

EXECUTION COUNT:

4.00000E+00

CURRENT RUN TIME:

9.00000E-05

MAXIMUM RUN TIME:

2.60000E-04

MINIMUM RUN TIME:

9.00000E-05

AVERAGE RUN TIME:

1.34500E-04

PLANNED RUN TIME:

4.50000E-05

TOTAL TIMING ERRORS:

4.00000E+00

EVAL

0.00000E+00

EXECUTION COUNT:

4.00000E+00

CURRENT RUN TIME:

1.15500E-03

MAXIMUM RUN TIME:

1.56500E-03

MINIMUM RUN TIME:

1.15000E-03

AVERAGE RUN TIME:

1.25850E-03

PLANNED RUN TIME:

9.92000E-04

TOTAL TIMING ERRORS:

4.00000E+00

SENSOR

0.00000E+00

EXECUTION COUNT:

5.00000E+00

CURRENT RUN TIME:

1.01000E-04

MAXIMUM RUN TIME:

2.60000E-04

MINIMUM RUN TIME:

9.00000E-05

AVERAGE RUN TIME:

1.27800E-04

PLANNED RUN TIME:

4.50000E-05

TOTAL TIMING ERRORS:

5.00000E+00

EVAL

0.00000E+00

EXECUTION COUNT:

5.00000E+00

CURRENT RUN TIME:

1.16000E-03

MAXIMUM RUN TIME:

1.56500E-03

MINIMUM RUN TIME:

1.15000E-03

AVERAGE RUN TIME:

1.23880E-03

PLANNED RUN TIME:

9.92000E-04

TOTAL TIMING ERRORS:

5.00000E+00

SENSOR

0.00000E+00

EXECUTION COUNT:

6.00000E+00

CURRENT RUN TIME:

9.10000E-05

MAXIMUM RUN TIME:

2.60000E-04

MINIMUM RUN TIME:

9.00000E-05

AVERAGE RUN TIME:

1.21667E-04

PLANNED RUN TIME:

4.50000E-05

TOTAL TIMING ERRORS:

6.00000E+00

EVAL

0.00000E+00

EXECUTION COUNT:

6.00000E+00

CURRENT RUN TIME:

1.13700E-03

MAXIMUM RUN TIME:

1.56500E-03

MINIMUM RUN TIME:

1.13700E-03

AVERAGE RUN TIME:

1.22183E-03

PLANNED RUN TIME:

9.92000E-04

TOTAL TIMING ERRORS:

6.00000E+00

=== 3RD RUN ===

SENSOR

0.00000E+00

EXECUTION COUNT:

1.00000E+00

CURRENT RUN TIME:

1.31000E-04

MAXIMUM RUN TIME:

1.31000E-04

MINIMUM RUN TIME:

1.31000E-04

AVERAGE RUN TIME:

1.31000E-04

PLANNED RUN TIME:

1.26000E-04

TOTAL TIMING ERRORS:

1.00000E+00

EVAL

0.00000E+00

EXECUTION COUNT:

1.00000E+00

CURRENT RUN TIME:

1.51100E-03

MAXIMUM RUN TIME:

1.51100E-03

MINIMUM RUN TIME:

1.51100E-03

AVERAGE RUN TIME:

1.51100E-03

PLANNED RUN TIME:

1.64300E-03

TOTAL TIMING ERRORS:

0.00000E+00

SENSOR

0.00000E+00

EXECUTION COUNT:

2.00000E+00

CURRENT RUN TIME:

9.40000E-05

MAXIMUM RUN TIME:

1.31000E-04

MINIMUM RUN TIME:

9.40000E-05

AVERAGE RUN TIME:

1.12500E-04

PLANNED RUN TIME:

1.26000E-04

TOTAL TIMING ERRORS:

1.00000E+00

EVAL

0.00000E+00

EXECUTION COUNT:

2.00000E+00

CURRENT RUN TIME:

1.25700E-03

MAXIMUM RUN TIME:

1.51100E-03

MINIMUM RUN TIME:

1.25700E-03

AVERAGE RUN TIME:

1.38400E-03

PLANNED RUN TIME:

1.64300E-03

TOTAL TIMING ERRORS:

0.00000E+00

SENSOR

0.00000E+00

EXECUTION COUNT:

3.00000E+00

CURRENT RUN TIME:

9.30000E-05

MAXIMUM RUN TIME:

1.31000E-04

MINIMUM RUN TIME:

9.30000E-05

AVERAGE RUN TIME:

1.06000E-04

PLANNED RUN TIME:

1.26000E-04

TOTAL TIMING ERRORS:

1.00000E+00

EVAL

0.00000E+00

EXECUTION COUNT:

3.00000E+00

CURRENT RUN TIME:

1.22100E-03

MAXIMUM RUN TIME:

1.51100E-03

MINIMUM RUN TIME:

1.22100E-03

AVERAGE RUN TIME:

1.32967E-03

PLANNED RUN TIME:

1.64300E-03

TOTAL TIMING ERRORS:

0.00000E+00

SENSOR

0.00000E+00

EXECUTION COUNT:

4.00000E+00

CURRENT RUN TIME:

9.10000E-05

MAXIMUM RUN TIME:

1.31000E-04

MINIMUM RUN TIME:

9.10000E-05

AVERAGE RUN TIME:

1.02250E-04

PLANNED RUN TIME:

1.26000E-04

TOTAL TIMING ERRORS:

1.00000E+00

EVAL

0.00000E+00

EXECUTION COUNT:

4.00000E+00

CURRENT RUN TIME:

1.23500E-03

MAXIMUM RUN TIME:

1.51100E-03

MINIMUM RUN TIME:

1.22100E-03

AVERAGE RUN TIME:

1.30600E-03

PLANNED RUN TIME:

1.64300E-03

TOTAL TIMING ERRORS:

0.00000E+00

SENSOR

0.00000E+00

EXECUTION COUNT:

5.00000E+00

CURRENT RUN TIME:

9.20000E-05

MAXIMUM RUN TIME:

1.31000E-04

MINIMUM RUN TIME:

9.10000E-05

AVERAGE RUN TIME:

1.00200E-04

PLANNED RUN TIME:

1.26000E-04

TOTAL TIMING ERRORS:

1.00000E+00

EVAL

0.00000E+00

EXECUTION COUNT:

5.00000E+00

CURRENT RUN TIME:

1.45800E-03

MAXIMUM RUN TIME:

1.51100E-03

MINIMUM RUN TIME:

1.22100E-03

AVERAGE RUN TIME:

1.33640E-03

PLANNED RUN TIME:

1.64300E-03

TOTAL TIMING ERRORS:

0.00000E+00

SENSOR

0.00000E+00

EXECUTION COUNT:

6.00000E+00

CURRENT RUN TIME:

9.20000E-05

MAXIMUM RUN TIME:

1.31000E-04

MINIMUM RUN TIME:

9.10000E-05

AVERAGE RUN TIME:

9.88333E-05

PLANNED RUN TIME:

1.26000E-04

TOTAL TIMING ERRORS:

1.00000E+00

EVAL

0.00000E+00

EXECUTION COUNT:

6.00000E+00

CURRENT RUN TIME:

1.24800E-03

MAXIMUM RUN TIME:

1.51100E-03

MINIMUM RUN TIME:

1.22100E-03

AVERAGE RUN TIME:

1.32167E-03

PLANNED RUN TIME:

1.64300E-03

TOTAL TIMING ERRORS:

0.00000E+00

SENSOR

0.00000E+00

EXECUTION COUNT:

7.00000E+00

CURRENT RUN TIME:

9.50000E-05

MAXIMUM RUN TIME:

1.31000E-04

MINIMUM RUN TIME:

9.10000E-05

AVERAGE RUN TIME:

9.82857E-05

PLANNED RUN TIME:

1.26000E-04

TOTAL TIMING ERRORS:

1.00000E+00

EVAL

0.00000E+00

EXECUTION COUNT:

7.00000E+00

CURRENT RUN TIME:

1.23500E-03

MAXIMUM RUN TIME:

1.51100E-03

MINIMUM RUN TIME:

1.22100E-03

AVERAGE RUN TIME:

1.30929E-03

PLANNED RUN TIME:

1.64300E-03

TOTAL TIMING ERRORS:

0.00000E+00

SENSOR

0.00000E+00

EXECUTION COUNT:

8.00000E+00

CURRENT RUN TIME:

9.50000E-05

MAXIMUM RUN TIME:

1.31000E-04

MINIMUM RUN TIME:

9.10000E-05

AVERAGE RUN TIME:

9.78750E-05

PLANNED RUN TIME:

1.26000E-04

TOTAL TIMING ERRORS:

1.00000E+00

EVAL

0.00000E+00

EXECUTION COUNT:

8.00000E+00

CURRENT RUN TIME:

1.24600E-03

MAXIMUM RUN TIME:

1.51100E-03

MINIMUM RUN TIME:

1.22100E-03

AVERAGE RUN TIME:

1.30137E-03

PLANNED RUN TIME:

1.64300E-03

TOTAL TIMING ERRORS:

0.00000E+00

SENSOR

0.00000E+00

EXECUTION COUNT:

9.00000E+00

CURRENT RUN TIME:

9.20000E-05

MAXIMUM RUN TIME:

1.31000E-04

MINIMUM RUN TIME:

9.10000E-05

AVERAGE RUN TIME:

9.72222E-05

PLANNED RUN TIME:

1.26000E-04

TOTAL TIMING ERRORS:

1.00000E+00

EVAL

0.00000E+00

EXECUTION COUNT:

9.00000E+00

CURRENT RUN TIME:

1.23300E-03

MAXIMUM RUN TIME:

1.51100E-03

MINIMUM RUN TIME:

1.22100E-03

AVERAGE RUN TIME:

1.29378E-03

PLANNED RUN TIME:

1.64300E-03

TOTAL TIMING ERRORS:

0.00000E+00

SENSOR

0.00000E+00

EXECUTION COUNT:

1.00000E+01

CURRENT RUN TIME:

9.30000E-05

MAXIMUM RUN TIME:

1.31000E-04

MINIMUM RUN TIME:

9.10000E-05

AVERAGE RUN TIME:

9.68000E-05

PLANNED RUN TIME:

1.26000E-04

TOTAL TIMING ERRORS:

1.00000E+00

EVAL

0.00000E+00

EXECUTION COUNT:

1.00000E+01

CURRENT RUN TIME:

1.23200E-03

MAXIMUM RUN TIME:

1.51100E-03

MINIMUM RUN TIME:

1.22100E-03

AVERAGE RUN TIME:

1.28760E-03

PLANNED RUN TIME:

1.64300E-03

TOTAL TIMING ERRORS:

0.00000E+00

=== END ===

GLOSSARY

Although there may be several definitions for the terms in this glossary, only those related to a term's usage in this thesis are included here.

Real-Time - events which occur on a scheduled basis in a timely manner.

Task set - grouping of predefined processes to be executed.

Run time - occurring during program execution.

Commercial Off The Shelf - equipment developed commercially and available on the open market.

Buffer - dedicated area in memory for storing data.

REFERENCES

1. Luqi, V. Berzins, *Rapidly Prototyping Real-Time Systems,* IEEE Software, September 1988, pp. 25-36.
2. Jeff England, et al., *Ada Performance Analyzer*, Technical Report, Intermetrics Inc, Huntington Beach, CA, 1985.
3. Gaurav Arora, David Stewart, *AFTER: A case tool to assist in Fine-tuning of embedded real-time systems,* University of Maryland, MD.
4. Thomas J. Ball, James R. Larus, *Optimally Profiling and Tracing Program,* TR 1031, Computer Sciences Department, University of Wisconsin-Madison, July 1991.
5. Vishal Jain, et al., *An approach for monitoring intrusion removal in Real Time Systems,* University of Pittsburgh and Trinity College, 1997.
6. M. van Riek, et al, *Monitoring of Distributed Memory Multicomputer Programs,* Laboratoire de l'Informatique du Parallelisme, and Department of Computer Science, University of Tennessee.
7. Luqi, M. Shing, *CAPS - A Tool for Real-Time System Development and Acquisition,* Quarterly Review, Office of Naval Research, Vol. XLIV, No. 1, 1992, pp. 12-16.
8. Luqi, *Handling Timing Constraints in Rapid Prototyping,* Proceedings of the Twenty-second annual Hawaii International Conference on System Science, 1989.
9. Luqi, V. Berzins, R. Yeh, *A Prototyping Language for Real-Time Software,* IEEE Transactions on Software Engineering, October 1988, Vol. 14, No. 10, pp. 1409-1423.
10. John Barnes, Ada 95 Rationale: The Language, The Standard Libraries. Lecture Notes in Computer Science, vol 1247, Springer-Verlag, 1997, ISBN 3-540-63143-7.
11. Luqi, M. Shing, *Real-Time Scheduling for a Software Prototyping,* Journal of Systems Integration, Vol. 6, 1996, pp. 41-72.
12. Awad. Maher, *OBJECT-ORIENTED TECHNOLOGY FOR REAL-TIME SYSTEMS: A Practical Approach Using OMT And Fusion,* Prentice Hall, NJ,

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center 2
   8725 John J. Kingman Road, Suite 0944
   Fort Belvoir, Virginia 22060-6218

2. Dudley Knox Library 2
   Naval Postgraduate School
   411 Dyer Rd.
   Montery, California 93943-5101

3. Center for Naval Analysis 1
   4401 Ford Avenue
   Alexandria, Virginia 22302-0268

4. Chief of Naval Research 1
   800 N. Qunicy Street
   Arlington, Virginia 22217

5. Dr. Man-Tak Shing, Code CS/Sh 1
   Naval Postgraduate School
   Montery, California 93943-5100

6. Dr. Ted Lewis, Code CS/Lt 1
   Chair, Computer Science Department
   Naval Postgraduate School
   Montery, California 93943-5100

7. Dr. Valdis Berzins, Code CS/Be 1
   Naval Postgraduate School
   Montery, California 93943-5100

8. Dr. Luqi, Code CS/Lq 1
   Naval Postgraduate School
   Montery, California 93943-5100

9. Library, Code D0274 1
   Naval Command. Control, and Ocean Surveillance Center
   RDT&E Division
   San Diego, California 92152-5001

10. Anh Le, Scientist, Code D0274 1
Naval Health Research Center
San Diego, California 92152-5001

11. Hon. John W. Douglas 1
Assistant Secretary of the Navy
(Inferences, Research, Development and Acquisition)
Room E741
1000 Navy Pentagon
Washington, DC 20350-1000

12. John Drummond, Scientist, Code D4123 1
Naval Command, Control, and Ocean Surveillance Center
RDT&E Division
San Diego, California 92152-5001

13. Dr. Marvin Langston 1
1225 Jefferson Davis Highway
Crystal Gateway 2 / Suite 1500
Arlington, Virginia 22202-4311

14. Capt. Talbot Manvel 1
Naval Sea Systems
2531 Jefferson Davis Hwy.
Arlington, Virginia 22240-5150

15. National Science Foundation 1
Attn: Bill Agresty
4201 Willson Blvd.
Arlington, Virginia 22230